Order Number 9236444

Object-oriented hierarchies across protection boundaries

Dykstra, David Wayne, Ph.D.

University of Illinois at Urbana-Champaign, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

OBJECT-ORIENTED HIERARCHIES ACROSS PROTECTION BOUNDARIES

BY

DAVID WAYNE DYKSTRA

B.S., University of Illinois, 1983
M.S., Illinois Institute of Technology, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1992

WE HEREBY RECOMMEND THAT THE THESIS BY

DAVID WAYNE DYKSTRA

ENTITLED OBJECT-ORIENTED HIERARCHIES ACROSS PROTECTION BOUNDARIES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

R H Campbell

Director of Thesis Research

M Faiman

Head of Department

Committee on Final Examination†

R H Campbell

Chairperson

† Required for doctor's degree but not for master's.

O-517

OBJECT-ORIENTED HIERARCHIES ACROSS PROTECTION BOUNDARIES

David Wayne Dykstra, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1992
Roy H. Campbell, Advisor

Protection is the mechanism employed by operating systems to control access to resources. Object encapsulation in object-based systems requires control of access to every object. The incremental definition of objects through inheritance and type hierarchies is an important aspect of object-oriented systems. This dissertation examines the relationship between protection and object-oriented hierarchies. Splitting object-oriented hierarchies across protection boundaries is particularly attractive for the purposes of providing a uniform programming model to object-oriented applications and for implementing a minimal object-oriented kernel.

After surveying current research and providing a background for discussion, this dissertation presents a detailed analysis of the issues relating to splitting object-oriented hierarchies across protection boundaries. The analysis is independent of language, operating system, and protection model. The analysis reveals the precautions that must be taken to guard against protection violations. The analysis also shows that in the general case an object must be able to be split across the protection boundaries, and that the child portion of the object should delegate or forward unrecognized method calls to the parent portion of the object on the other side of the boundary.

A practical implementation of object-oriented hierarchies across protection boundaries is presented. The implementation uses C++ and the *Choices* object-oriented operating system. The implementation is based on proxy objects and a partitioned rings of protection model. Proxy objects are automatically allocated, validated, and stripped off to provide an interface that is transparent to the programmer. A tool called Proxify++ assists in providing necessary information to the run-time system. Examples of the use of the implementation are provided, and experience gained by moving the filesystem hierarchy outside of the kernel is presented. Performance of the implementation is also evaluated, and calls to the kernel are found to be comparable to operating systems that are not object-oriented. Performance for calls to intermediate rings is found to be superior to calls to separate address spaces.

Dedication

To my wife Ruth, for going through the trials of graduate school with me.
And to my dog Tuppence and cat Amanda for providing moral support
by sleeping by me as I wrote:
Tuppence at my feet and Amanda on my lap.

Acknowledgements

I thank my advisor Professor Roy Campbell for providing me with the opportunity to do this research. I also thank Professor Ralph Johnson for taking a particular interest in my work, as well as the other members of my committee: Professors Jane Liu, Andrew Chien, and Tony P. Ng.

I thank Vince Russo and Peter Madany for writing most of the rest of *Choices* on which my work is based and for helping me with portions of my design. I also acknowledge other students with whom I have worked on *Choices* including but not limited to Gary Johnston, Aamod Sane, See-Mong Tan, Bjorn Helgaas, Gary Murakami, John Coolidge, and Johnny Zweig.

I thank AT&T Teletype for supporting me as I went back to school full-time and allowing me to work part-time for them remotely, and I thank Memorex-Telex for continuing that support later. I also thank AT&T Bell Labs for giving me some time to work on my thesis after I went back to work full-time.

I thank the many friends that I made while in Champaign, mostly from Hessel Park Church and the Illini Squares square dance club, for encouraging me while I was in school.

Finally, I am most grateful to my wife Ruth for her love and encouragement.

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

In the context of programming systems, *protection* is the mechanism used to control access of programs and users to resources, including data [PS85]. Protection is often introduced to ensure integrity, to prevent inappropriate use that would intentionally or unintentionally compromise a resource access policy that is beneficial to the users of a system. Protection may also be used to ensure security, but this is a broader topic than the issues addressed in this project. My research is concerned with protection and the design of an operating system that supports an object-oriented interface to applications. This work examines the relationship between protection and object-oriented systems.

Protection can be enforced by *convention* or agreement amongst the users of the system to program in a specific manner, by a *compiler* that statically checks protection rules are not violated or inserts run time checks to verify that the access rules are not violated at execution time, or by *hardware* that absolutely prevents violation of access rules at execution time. Protection by convention can easily be violated. Similarly, protection by compilers depends on the effectiveness and correctness of the compiler, the nature of the language, and whether multiple languages are used to build applications. Hardware-enforced protection offers more integrity for access policies and can be applied to more general access problems.

In object-oriented systems, *encapsulation* [Nie89] is the major programming protection mechanism that restricts manipulation of objects defined in the representation of an abstract data type from being accessed except by a set of predefined methods or operations. Encapsulation is beneficial because it reduces coupling between modules, encouraging modification,

reuse, porting, and understanding. Encapsulation can be enforced in the same ways as general protection: by convention as in languages that do not have features that are object-oriented, by compiler as in Smalltalk [GR83, Gol84], or by hardware, as in various capability-based computer architectures. Where capabilities are not built into a hardware architecture, hardware protection may also be provided by a combination of hardware and operating system primitives. For example, the supervisor/user state distinction or virtual memory mechanism may be used to provide hardware-enforced encapsulation.

Encapsulation is, however, only one aspect of an object-oriented system. As defined by [Weg87], objects in object-oriented systems belong to classes and class hierarchies that can be incrementally defined by an inheritance mechanism. These hierarchies are an important aspect of object-oriented systems and greatly facilitate reuse through the development of frameworks [JR91]. This research examines extending these hierarchies across protection boundaries.

1.1 Motivation

An object-oriented operating system, like any object-oriented program, is made up of many object-oriented hierarchies. There are two important motivations to extend these object-oriented hierarchies across the protection boundaries that an operating system places around itself:

1. This extension will provide applications which are based on an object-oriented programming paradigm with a uniform programming model as they interface with the operating system. It will enable the applications to interface with objects, methods, and inheritance rather than the traditional procedure call.

2. The ability to split object-oriented hierarchies across protection boundaries is attractive as a mechanism to structure the operating system itself into more-privileged and less-privileged portions. Keeping the most highly privileged kernel of an operating system to a minimum has been proven to be a good way to structure an operating system. Allowing object-oriented hierarchies to extend out of the protected kernel will make it easier to move portions out of or into the kernel. It will also make a uniform programming model as the portions of the operating system outside of the kernel interface with the kernel, and as the kernel interfaces with the rest of the system.

An object-oriented hierarchy can be split across a protection boundary by either allowing parent classes to operate on both sides of the boundary or splitting up the parent and child classes in the hierarchy so that they operate on different sides of the boundary. For example, a general purpose class such as a class that defines linked lists may be defined inside an operating system; if an application makes a subclass of that class it is simplist to copy the parent class so the parent operates directly in the application rather than requiring protection boundary crossings for inherited behavior. On the other hand, a class that implements an operating system service such as a sémaphore must remain protected when an application makes a subclass because the service will need to have access to protected shared operating system data structures. This second kind of hierarchy split is a harder problem than the first. Both kinds are useful.

Splitting object-oriented hierarchies across protection boundaries has not been significantly researched before this work. Protection systems in the past have either been designed using the function call paradigm as an interface or using the object-based paradigm with no inheritance hierarchies. Research is needed to discover the relationship between protection and object-oriented hierarchies.

1.2 Thesis

The focus of this research is the development and study of a mechanism to support object-oriented hierarchies across protection boundaries. The mechanism should strictly enforce protection and yet be as transparent as possible to the programmer to allow a uniform object-oriented programming model. In addition, the mechanism must have adequate performance so as to not preclude its use.

The mechanism that I designed and implemented fulfills these requirements. Applications not only can interface with system objects as if they are local objects, they can also incrementally modify the behavior of the system objects for their own use through inheritance. The operating system can also take advantage of the inclusion polymorphism of type hierarchies to define interfaces through which it can call out to less-privileged subtypes. The mechanism automatically and transparently inserts and removes special objects called proxy objects which are used to cross a protection boundary and to invoke methods on individual objects on the

other side of the boundary. The performance of crossing protection boundaries is adequate: method invocations into the kernel with no parameters takes approximately the same amount of time as a Unix[1] system call, and the extra time taken for allocation and removal of proxy objects for parameters and return values is not detrimental because those operations only occur on a relatively small percentage of calls in a typical application. My implementation also uses intermediate rings to support structuring of the operating system into kernel and non-kernel portions; calls to the intermediate rings take more time than calls to the kernel, but they are faster than calls to separate address spaces because message passing is not required. The implementation is for the C++ object-oriented language [Str86, Jor90] and the *Choices* object-oriented operating system [CJR87, CRJ87, CJMR89, Rus91, RMC90].

The study of this mechanism has led to an analysis of the fundamental problems caused by splitting type hierarchies and inheritance hierarchies across protection boundaries. The analysis is independent of language, system, and protection model. The analysis reveals the precautions that must be taken to guard against protection violations in an object-oriented system with protection boundaries. The analysis also shows that in the general case an object must be able to be split across the protection boundaries, and that the portion of the object belonging to a child in an object-oriented hierarchy should delegate or preferably forward unrecognized method calls to the parent portion of the object on the other side of the boundary.

1.3 Motivating Examples

In this section I propose a few motivating examples for the desire to split object-oriented hierarchies across protection boundaries. While developing the *Choices* object-oriented operating system, other developers and I have encountered several categories of situations in which it would be helpful to extend the object-oriented hierarchies out of a protected kernel.

1.3.1 Providing System Services

The first category involves providing system services to object-oriented applications in an object-oriented manner. For example, if the operating system provides a Process class to represent processes, user programs should be allowed to invoke methods on a Process object as easily as

[1] Unix is a trademark of AT&T.

the kernel itself can. This example does not show an extension of an object-oriented hierarchy across a protection boundary, but it does show the use of an object across a boundary.

1.3.2 Customizing Services at User Level

The next category involves customizing operating system services at the user level. For example, there is an OutputStream type in *Choices* that has a write() method to send data to the stream and a flush() method that is called when data is to be written out. We would like to be able to provide a BufferedOutputStream in user space that overloads the write() and flush() methods to save data in a local buffer and reduces the number of times that system calls need to be made. The BufferedOutputStream should be a subtype of OutputStream so the user can treat it as an OutputStream through the inclusion polymorphism of the type hierarchy. This example also shows an extension of the implementation hierarchy in that BufferedOutputStream inherits some methods from its parent OutputStream. The BufferedOutputStream example is best implemented by copying the code for the parent OutputStream class into the application because the parent class code does not need to cause any side effects in the operating system.

We would also like to be able to provide applications with the ability to make a subclass of a class that implements an operating system service and cannot be copied to an application because it needs to continue to cause side effects in the operating system. For example, the code for the Semaphore class cannot be copied into an application because it needs to be able to modify protected operating system data that is shared between applications. We would like to be able to provide a UserSemaphore subclass of Semaphore that redefines methods to avoid system calls if a resource is known to be free, and only calls the parent methods in the operating system when necessary.

1.3.3 Customizing Services at an Intermediate Level

Another category involves having a minimal kernel and moving many shared system services into an intermediate protection level. The kernel will be protected from the intermediate system services, and the intermediate system services will be protected from user programs. There are many advantages to a minimal kernel as will be discussed in section 2.5.

When there are more than two levels of protection, there will be cases where a type is defined on one level, a subtype is defined on another level, and the type or subtype is used on

a third level. For example, there is a Timer type in *Choices* that defines start() and await() methods. We want to define the Timer type at the minimal kernel level, define a subtype called TimeoutTimer at an intermediate level, and allow the user level to use the TimeoutTimer type and treat it as a Timer through inclusion polymorphism.

1.3.4 Separating Policy from Mechanism

The final category is that of providing a mechanism at a protected lower level and a policy at a less-protected higher level. For example, we want to allow higher levels to control the policy of handling page faults. The kernel will define an interface type called FaultHandler and define methods implementing the default policy for handling a page fault. User programs will be able to make a subtype of FaultHandler and request the kernel to invoke methods on the subtype through inclusion polymorphism.

This same kind of feature could be provided without using subtypes, but using subtypes and polymorphism takes advantage of the object-oriented programming paradigm.

1.4 Overview

The remainder of this dissertation is organized as follows. Chapter 2 gives background and definitions for following discussions and explores related work. Chapter 3 presents the analysis of splitting object-oriented hierarchies across protection boundaries. Chapter 4 is a high-level description of the implementation of object-oriented hierarchies across protection boundaries in the *Choices* operating system. Chapter 5 presents simple examples of split hierarchies in this system. Chapter 6 discusses experience in moving the *Choices* filesystem out of the kernel. Chapter 7 describes specific low-level details of the implementation, intended as a guide for those who desire to completely understand and modify the implementation. Chapter 8 discusses performance implications. Finally, chapter 9 draws conclusions from this work and proposes further research.

Chapter 2

Background

This chapter contains background information for the rest of this dissertation by describing the notation that I use, by defining terms, by discussing concepts of protection, operating systems, and object-oriented programming, and by exploring the similarities and differences between my research and related work.

These are the purposes of this chapter:

1. To provide background for understanding my analysis and implementation of object-oriented hierarchies across protection boundaries.
2. To show the previous work from which my research has benefitted.
3. To show that research on the topic of object-oriented hierarchies across protection boundaries is needed because the topic has not been significantly researched before this work.

2.1 Notation

These are the fonts that will be used in this dissertation and their meanings:

Italics - for defining terms and for the name *Choices*.

Bold - for language keywords and access rights.

Sans serif - for class, method, function, and variable names. Methods and functions are followed by a pair of parentheses ().

`Teletype` - for instances of the named class.

CAPITALS - for macros and defines. Macros are followed by a pair of parentheses ().

2.2 Object-Oriented Operating System Protection

Protection in object-oriented operating systems is defined in terms of objects, subjects, protection domains, and processes [PS85].

object - A unit of data. In object-oriented systems, the term takes on additional meaning: objects are *encapsulated* such that only the classes that "own" them may manipulate them. Some people define objects in object-oriented systems to also include the operations (code) with which they are encapsulated, but for the purposes of discussing protection in this dissertation, operations will not be considered to be part of an object.

subject - A unit of code. In object-oriented systems, the implementation of the operations (methods) of each class is a subject.

protection domain - A list of access rights to objects. Each subject is associated with one protection domain that specifies the access rights that the subject has to objects. There can be more than one subject and/or object for each protection domain, and objects can be in one or more protection domains. Multiple access lists are often viewed together in matrix form; such a matrix is called an *access matrix.* In this work, the access rights that I am concerned with are the ability to modify objects (that is, both **read** and **write** access rights together) and the ability to execute methods on objects (the **execute** access right).

process - A thread of control. Processes wind their way through different subjects. Subjects can do nothing without a process to activate them. Processes change protection domains as they change from one subject to another. Changing from one protection domain to another is called crossing a *protection boundary.* This occurs at the time a process calls a method on an object that belongs to a subject in a different protection domain from the one it was executing in, assuming that the process has the **execute** right on that object.

After the process crosses the protection boundary, its rights are *amplified* such that the new protection domain includes the right to modify the object. (The notion of amplification was first introduced in Hydra [Wul81].) The new protection domain may also contain new rights to other objects, but the called method receives a pointer to the object as a parameter so it can restrict itself to only modify that object. When the method is finished, the process returns back across the protection boundary and reverts to the protection domain without the amplified rights.

At any given time, a process should be allowed to access only the objects that it needs to get its job done. This is called the *need-to-know* principle, and it is useful in limiting the amount of damage that flawed code can cause.

2.3 Trust

In an operating system environment, different kinds of *trust* relationships exist between subjects. For example, subjects in a kernel are more trusted than subjects in applications, and subjects in different applications do not usually trust each other.

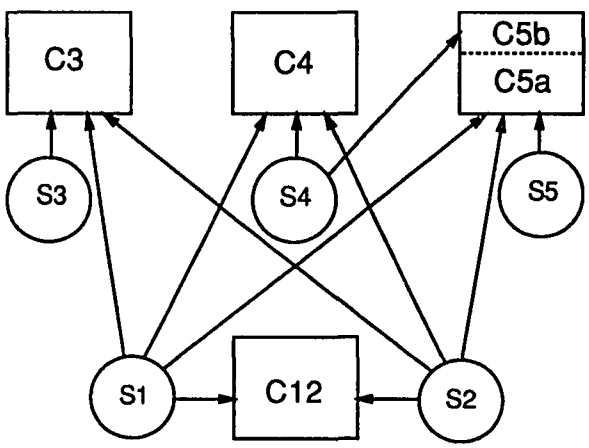

Figure 2.1: Trust Relationships

Figure 2.1 illustrates the trust relationships that can exist between subjects. Circles represent subjects, boxes represent collections of objects, and arrows indicate **read** and **write** access rights to an object in the protection domain of a subject. Table 2.3 shows the same information

in the form of an access matrix for the protection domains; D12 is the protection domain of S1 and S2, D3 is the domain of S3, D4 is the domain of S4, and D5 is the domain of S5.

	C12	C3	C4	C5a	C5b
D12	**read-write**	**read-write**	**read-write**	**read-write**	**read-write**
D3		**read-write**			
D4			**read-write**		**read-write**
D5				**read-write**	**read-write**

Table 2.1: Access Matrix for Trust Relationships

These are the trust relationships between two subjects:

more trusted - The more trusted subject is able to modify all objects belonging to the less trusted subject. The protection domain of the more trusted subject is a superset of the protection domain of the other subject. S1 and S2 are more trusted than the other subjects in the figure. They could be two parts of the kernel of an operating system.

less trusted - The converse of more trusted: the protection domain of the less trusted subject is a subset of the protection domain of the other subject.

partially trusted - The partially trusted subject is able to modify some of the objects that belong to the other subject but not all. S5 partially trusts S4; S4 has access to C5b but not C5a. They could be two applications that are working together and have some shared virtual memory.

mutually distrusted - Mutually distrusted subjects do not trust each other with any of the other's objects. S3 and S4 mutually distrust each other. They could be two independent applications.

mutually trusted - Mutually trusted subjects trust each other with all of each other's objects. They both have the same protection domain. S1 and S2 mutually trust each other with their collection of objects in C12.

When any two subjects do not mutually trust each other, a protection boundary is placed between them so that when a process crosses from one subject to the other the process will

have the proper access rights (that is, the process will be in the proper protection domain). When one subject is more trusted than another subject, the protection boundary prevents the less trusted subject from modifying the objects belonging to the more trusted subject; with this kind of a trust relationship, the two subjects are at different *protection levels* because one is more protected than the other.

Even though trusted subjects can modify the objects that belong to subjects that trust them, that does not mean they will modify the objects. Trust simply implies that one subject is willing to accept that the other subject will abide by any compile-time or run-time enforcement of object encapsulation.

These trust relationships exist regardless of how fine-grained or coarse-grained the protection mechanism is. That is, whether the objects are individually protected through capability-based hardware or protected as a group through some other means, the analysis of chapter 3 still applies.

2.4 Proxies

My work makes extensive use of proxy objects. *Proxy objects* are objects that represent other objects. All interactions with objects that are not in the same protection domain or address space of the user go through proxy objects. The use of the term in object-oriented systems was first introduced in [Sha86] with the following "proxy principle": "In order to use some service, a potential client must first acquire a proxy for this service; the proxy is the only visible interface to the service." Proxies forward calls to the real objects and are indistinguishable from the real objects from the client's point of view.

Proxies in *Choices* (first introduced in [Rus91]) are a specialization of the general proxy concept. Proxies in the general case can represent any number of objects, and the objects can be distributed onto different machines. They can also be independently implemented for each different kind of server object. Systems that use these general proxies are SOS [SGM89], Comandos [MG89], and HCS [Not87]. In my work in *Choices*, proxies represent only one object on the same machine, and they are identically implemented no matter what kind of object they represent.

The proxy concept was extended by SOS to include a concept of Fragmented Objects [SGH+89]. In that system an individual logical object can be distributed across multiple machines, with a fragment of the logical object on each machine. My system also supports fragmenting a logical object into two or more parts: when a class is subclassed outside of the protection domain it is defined in, the portions of the object belonging to the parent and child in the different protection domains are fragmented into those different domains. SOS considers a proxy to be a fragment of the object that it represents, but in my system proxies are considered to be only the mechanism (along with inherited method stubs) that ties the fragments of the object together so they appear to the client to be a single object.

2.5 Minimal Kernel

A primary motivation for my research is to assist in creating a minimal kernel. A minimal kernel, or microkernel, contains only the essential system services necessary to convert bare hardware into a small number of abstractions. Any services that can be performed outside of the kernel without significantly degrading performance should not be a part of the kernel. Minimal kernels typically provide basic services such as process management, memory management, and inter-process communication.

A minimal kernel has several advantages over a large, monolithic kernel:

1. Since a minimal kernel is smaller, it is easier to develop, debug, and maintain. Debugging a kernel can be very difficult because all of the code executes at the highest privilege level, and errors often crash the entire system. The more complex the kernel, the more chance that serious errors will occur.

2. A minimal kernel encourages the separation of policy and mechanism. Separation of policy and mechanism is very important to achieve modularity and flexibility in an operating system [WCC+74]. The kernel provides the mechanism, and the policies are implemented outside of the kernel.

3. Since most of the operating system services are implemented outside of a minimal kernel, those services can easily be updated or new services can be added without the need to

recompile the kernel or reboot the system and without jeopardizing the integrity of the entire system. This is especially valuable for embedded systems that require high uptime.

4. A minimal kernel makes it possible to run variations of the same operating system or totally unrelated operating systems on the same machine. This is especially valuable for a research environment where the operating system changes often and different programmers have their own versions of the operating system.

Many other modern operating systems incorporate the minimal kernel concept; examples include Mach [R+89], V [CZ83], Amoeba [TM86, MvRT+90], and a kernel for Clouds called Ra [BA+89]. None of these systems are object-oriented, but the advantages of a minimal kernel are still applicable to object-oriented systems. Mach, V, and Amoeba are message-based systems. Even though the Ra kernel is written in C++, it is only object-based outside of the kernel, not object-oriented.

2.6 Protection Model

My implementation uses a combination of a rings of protection model and a capabilities model. Both of these models have been widely used in other systems. The combination of the two models fits well with introducing protection boundaries into object-oriented languages that are based on one large shared address space.

2.6.1 Protection Rings

In a system that incorporates protection rings, each process executes in a protection ring that determines which objects it can access. The innermost ring has the greatest access rights. Each successive ring restricts access further. A process can only enter into further-in rings through controlled access points.

MULTICS [Org80] was the first system to use protection rings. MULTICS employs a large number of protection rings (32) and assigns specific services to each ring. The disadvantage of a large number of rings is that it becomes difficult to hierarchically order services; the interdependencies between services can get in the way. My implementation can accommodate any number of rings, but I anticipate that only a very small number will be used, perhaps three

or four. The most essential portions will remain in the kernel ring and the remainder will be placed in one or two rings outside of that with applications in the outermost ring.

VAX/VMS [KB84] does not use the ring terminology, but it employs four protection levels which are essentially the same as rings. In addition, the kernel address space is shared between all applications. My system also shares address space; the address space of each inner ring is shared with all further-out rings. The address space of further-in rings is shared but is not accessible to the further-out rings. The advantage of the sharing is that, when a process crosses into a further-in ring, then the address space of the ring that the process came from is directly accessible for manipulation by the inner ring.

My system also includes the ability to partition any ring other than the kernel into separate independent address spaces. It is common practice to be able to partition the outermost (application) ring into separate address spaces, but the ability to flexibly partition inner rings is novel to this work.

2.6.2 Capabilities

Capabilities [Lev84, BS88, Dei84] are protected pointers that provide rights to access individual objects. A proxy in my system is a capability that provides a partition of one ring the right to invoke methods on an individual object in another ring.

Capability-based systems typically use fine-grained protection and require every access to every object to go through a capability. The problem with that is that the management of the capabilities becomes far too complex, and special-purpose hardware is required for efficient implementation. Even with special-purpose hardware there is overhead to check the protection on every access. My implementation avoids much of that overhead by grouping collections of objects together into a partition of a ring using conventional virtual memory hardware. Once a process has gained access to an object in another partition of another ring through a proxy, the process may then go on to access any other object in that partition without going through another proxy.

Many systems have been developed that incorporate the capabilities concept of protecting objects separately. Many of them provide a separate address space for every object using conventional virtual memory hardware. Examples include Amoeba [TM86], Clouds [DLA88, PD88], COOL [HM90] (based on Chorus) and Eden [ABLN85]. Using conventional virtual

memory hardware to protect each object individually is not well suited for small variably-sized objects because every object must take up a multiple of a fixed page size. Also, separate page tables need to be maintained for every object. Eden goes further and uses active objects, where each object has a process associated with it as well as an address space; communication is done via message passing. My system uses passive objects that are manipulated through method calls by independent processes.

BiiN [PJC+90, PKD+90] is a modern system that incorporated capability-based hardware. In the BiiN architecture, the capabilities implemented by the hardware are called *access descriptors*. Objects can be of any size from 64 bytes to 4 gigabytes, and up to 2^{26} objects can be actively addressed by a process. This hardware virtually eliminates the problems of wasted space and time that plague systems that protect individual objects with conventional hardware. However, funding was removed from BiiN in late 1989 and interest in capability-based hardware has been fading since.

None of these systems are truly object-oriented, they are all object-based. In other words, they do not incorporate inheritance and class hierarchies. If they were to extend their systems to incorporate inheritance, they would have to deal with most of the same issues that I address in my research.

2.7 Extending Object-Oriented Operating Systems

There are two systems in the literature that extend object-oriented hierarchies beyond an object-oriented operating system.

Comandos [MG89, GM89] is one such system. A major goal of Comandos is to provide an integrated computational model for programmers of object-oriented applications as they interact with the operating system. The designers of Comandos intend to provide support for multiple object-oriented programming languages. Inheritance is supported by having an implementation contain only the subset of operations defined by a type, and by keeping the operations defined by the type's supertype in a different implementation object. My system does this also. Few details of how Comandos does the split are provided in the literature, however. Work has proceeded in the development of a language called OSCAR that transparently takes advantage of the Comandos inheritance structure; Guide [KMV+90] is a language that is an

exploratory implementation of some of the OSCAR features. Comandos is not a self-contained system, it runs on top of other systems: a minimal kernel called IK, and Unix.

Muse [YTT89, YTM+91] is the other system. Muse has a reflective architecture that is designed to be easily modifiable and optimizable for the needs of applications. Muse is based on a meta-object hierarchy initially with three levels: the object level is the equivalent of application level, the meta-object level is an intermediate level, and the meta-meta-object level is the equivalent of kernel level. Each object is active (that is, has its own process) and is rather large-grained in its own address space. A delegation mechanism implements an object-oriented hierarchy between the objects, but few details are provided and examples only show delegation between objects in the meta-object layer.

Both of these systems have only briefly addressed the problems of object-oriented hierarchies across protection boundaries. These systems show that there is interest in the topic, but more research is needed.

2.8 Extending C++

Several systems in the literature extend the C++ language to support more than a single, large, statically compiled address space.

Extended C++ [Sel90] supports remote procedure calls in C++. It is implemented as a translator that translates a superset of C++ into ordinary C++ on top of Unix. It adds a **remotable** keyword to mark methods that can be called remotely. This is analogous to the **proxiable** keyword in my system which marks methods that can be called from outside of a compiled module. Extended C++ extends the language further to bundle together parameters to be sent along with the remote procedure call, including hand-written encoders and decoders for parameters that have a complex structure. That is not necessary in my system because called methods have direct access to the address space of the caller if the caller trusts the called method.

The SOS system [SGM89] permits migration of objects from one machine to another in a distributed system. SOS is implemented with a modified C++ compiler and runs on top of Unix. The modified C++ compiler supports a **dynamic** keyword that is used to instantiate an object that can migrate across address spaces. Rather than having hand-written encoders and

decoders for parameters like Extended C++, SOS uses a hand-written proxy for each object type which is specialized for the methods and parameters that the object accepts.

A system that allows adding code to a running C++ program is described in [DSS90]. That system uses first-class classes (objects that represent classes at run-time [IL90]) to keep track of C++ class hierarchies, as does my system. However, that system does not provide any protection between already running parts and newly loaded parts. The system also runs on top of Unix.

2.9 Implementation Hierarchies and Type Hierarchies

There are two kinds of hierarchies in object-oriented systems: implementation hierarchies and type hierarchies [Lis87]. *Implementation hierarchies* are hierarchies in which code and data implementations are shared through inheritance. A *child* in an implementation hierarchy contains all of the methods and data of its *parent(s)* but can replace some of the methods and can add new methods and new data.

By contrast, *type hierarchies* are hierarchies in which calling signatures and semantic meanings of methods are shared. Calling *signatures* are the method names, parameters, and return values that are supported by a type. A child (subtype) in a type hierarchy has methods with the same names, parameters, and return values as its parent(s) (supertype(s)), and each of the matching methods perform similar operations. A child can also have additional methods.

Type hierarchies are used for *inclusion polymorphism* [CW85]: where a particular set of operations is required, a type that has those operations or any of its children in the type hierarchy can be employed.

In object-oriented languages that have the concept of classes, the implementation hierarchy follows the class hierarchy. In some of those languages the type hierarchy also always follows the class hierarchy, but in other languages the type hierarchy can be independent of the class hierarchy. An example of the former is C++, and an example of the latter is Smalltalk because it employs dynamic type checking[1]. In any case, the distinction is a useful one and I will use it in the analysis.

[1] Smalltalk does not enforce any hierarchy on types. Instead, every method invocation is handled independently and the object responds if it has defined a method of that name. This is known as the *grouping* approach [Lis87]. Nevertheless, a type hierarchy inherently exists.

2.10 Inheritance, Delegation, and Forwarding

Inheritance and delegation are two ways of looking at object-oriented implementation hierarchies. The *Treaty of Orlando* [LSU87] distinguishes between inheritance and delegation; when I use the two terms in this dissertation, the distinguishing property is whether the sharing in the implementation hierarchy is *static* or *dynamic*. That is, *inheritance* specifies that the sharing pattern is statically determined by the time an object is created, and *delegation* allows the sharing to be determined dynamically during run-time. C++ and Smalltalk are inheritance-based languages, and Self [US87, CUL89] is a delegation-based language.

The way that delegation-based languages provide dynamic determination of the sharing pattern is by giving each level of the implementation hierarchy a separate object, and *forwarding* unrecognized methods to the parent object. The parent object can be changed at any time by choosing to forward methods to a different object.

True delegation is more than just forwarding a method call to the parent object [Lie86]. In true delegation (and inheritance), if a child overloads a method that its parent invokes, the child's version of the method will be used. That is, the method gets invoked on the original object. On the other hand, if method calls are simply forwarded to the parent object, the parent would only invoke its own version of the method. Figure 2.2 pictures the distinction between delegation and forwarding.

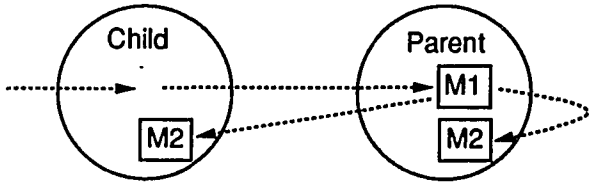

Figure 2.2: Delegation and Forwarding

Suppose a parent defines a method M1 that invokes another of its methods M2, and its child redefines method M2. When a call to M1 comes to the child, it will forward the call to its parent because the child does not have an M1. With simple forwarding, the parent will invoke its own M2, but with delegation, the parent will invoke the child's M2. Delegation-based languages implement this by sending a reference to the child object ("self") along with the method calls

that are delegated to the parent object, and by always beginning with the child object when invoking methods.

2.11 Summary

This chapter has provided background for understanding my analysis and implementation of object-oriented hierarchies across protection boundaries. It has also shown the previous work from which my research has benefitted. Most importantly, it has shown that research on the topic of object-oriented hierarchies across protection boundaries is needed because the topic has not been significantly researched before this work.

Chapter 3

Analysis

The intention of the analysis in this chapter is to expose the general principles that I have learned by implementing a system that supports object-oriented hierarchies across protection boundaries. These general principles are applicable to any object-oriented language, any operating system, and any protection model.

The analysis begins by examining objects protected by protection boundaries, moves on to type hierarchies split across protection boundaries, and finishes with implementation hierarchies split across protection boundaries. A summary of the principles learned is at the end of the chapter.

3.1 Using Objects Protected by Boundaries

Before examining the splitting of object-oriented hierarchies across protection boundaries, I will first address the concerns caused by using objects that are protected by protection boundaries. For example, an operating system may contain many objects that are of various types in a type hierarchy; how should the operating system protect itself while providing services? In other words, what precautions need to be taken when an untrusted subject calls a method on an object? Precautions are necessary to prevent the untrusted subject from circumventing protection by taking advantage of the amplified access rights in the protection domain of the called method.

3.1.1 Method Calls on Selected Objects

The first principle is that an untrusted subject should only be able to invoke methods on selected objects; that is, the protection domain of the subject must have the **execute** right for those selected objects. This follows directly from the need-to-know principle. Since calls to methods in different protection domains first cross over to the object's protection domain, untrusted subjects should not be allowed to call methods on all objects in a system. A subject will want to selectively give away the right for untrusted subjects to execute methods on its objects. For example, referring back to figure 2.1, suppose there are two untrusted subjects S3 and S4 that mutually distrust each other and a trusted subject S1 that is more trusted than both. In that case, S1 might give away the right to execute methods on one object to S3 and give away the right to execute methods on a different object to S4, but not give away the right to execute the same object to both untrusted subjects. Also, S1 may have some objects that should not be accessible by any untrusted subject.

3.1.2 Selected Method Calls on Objects

The second principle is that an untrusted subject should only be allowed to invoke selected methods on the objects that are accessible across protection boundaries. This also follows from the need-to-know principle. Programmers often want to have methods that are only used for internal purposes. Also, methods that are callable across protection boundaries must be written with the awareness that callers are not to be trusted: for example, parameters may need to be checked to ensure that they are within a valid range.

Thus, a subject should be able to give away selectively access rights to execute specific methods when it gives away **execute** rights on its objects. If a special mechanism does not exist, a programmer could introduce an intermediate object that only exposes the methods to be callable across boundaries and forwards method calls to the real object, but that is inconvenient.

3.1.3 Parameters Referring to Other Objects

Some methods accept parameters that refer to other objects. Presumably the methods will use the references to invoke methods on the other objects. An untrusted caller must therefore not be allowed to pass in a reference to an object that the caller has no right to access. For

example, some methods in *Choices* accept parameters that refer to `OutputStream` objects, and then invoke the write() method on those objects to write out messages to output devices or files. An untrusted caller must not be permitted to pass in a reference to an `OutputStream` object that can write to a privileged file into which the caller is not authorized to write.

In addition, an untrusted subject must not be permitted to pass in an object of an incorrect type. A correct type is either the type that the called subject expects or a subtype in the type hierarchy. If an object of an incorrect type were allowed to be passed through to the called subject, the methods that the called subject expects to be available on the object would not be available. For example, if a method was expecting an object of type OutputStream and the untrusted caller instead tried to pass in an object of type InputStream, the write() method would not be available on the object. In a language that has no run-time type checking (such as C++), this could cause severe consequences; it could even cause the system to crash. Run-time type checking is required for parameters from untrusted callers.

3.1.4 Creation and Deletion of Objects by Untrusted Subjects

The next question is whether a subject should be able to create or delete objects belonging to subjects that do not trust it. The creation and deletion of objects must be under the control of the subjects that own them, but a subject should be able to selectively give away the right for untrusted subjects to create its objects and to delete objects so created. Often objects exist to provide services for the untrusted subjects, and the untrusted subjects need to be able to request more services or relinquish services.

3.2 Type Hierarchy Split Across Protection Boundaries

Object-oriented systems use the type hierarchy to provide inclusion polymorphism. Subtypes in a hierarchy can transparently take the place of supertypes. Additional concerns are raised when a subtype and a supertype in the type hierarchy are defined on different sides of a protection boundary.

Consider first the case where the caller of a method expects a type that is defined on the other side of a protection boundary but instead a subtype is substituted on the same side as the caller. For example, suppose an operating system defines a supertype called Semaphore

and an application defines a subtype called LoggingSemaphore that redefines the methods of the supertype, and then the application treats a **LoggingSemaphore** object as if it were a **Semaphore** object. This does not cause any problems; a call to one of those methods will simply not cross the protection boundary.

On the other hand, consider the case where the substitution of a subtype causes an unexpected protection boundary crossing to an untrusted subject. For example, suppose an operating system was expecting an object of type Semaphore and instead the object was of type LoggingSemaphore and defined by an application. In some ways this is like a trojan horse, with an untrusted object masquerading as a trusted object. This may be useful in some circumstances, but in general such substitutions of untrusted subtype objects should be restricted for two reasons:

1. Parameters need to be chosen with care. Parameters to a call to an untrusted subject are like return values from a call from an untrusted subject. If a parameter refers to another object, passing a reference to that object can give away access rights to that object, and that should only be done if the caller intends to give away the rights.

2. Return values need to be restricted. Return values from a call to an untrusted subject are like parameters to a call from an untrusted subject. Values may need to be range-checked, and return values that refer to other objects need to be verified for accessibility and type as discussed in section 3.1.3.

3.3 Implementation Hierarchy Across Protection Boundaries

What happens when an implementation hierarchy is split across protection boundaries? Not only code is shared in an object-oriented implementation hierarchy. Each level of the hierarchy may also add data items (member variables) to the definition of objects. The data items added for a particular level are primarily intended to be accessed and modified by the code for that level of the hierarchy, although some languages make exceptions and allow code further down the hierarchy to modify the data. (C++ even goes so far as to allow parts of some objects, the **public** parts, to be modified by any subject.)

3.3.1 Unified or Split Objects

Should the portions of an object belonging to the different levels in the implementation hierarchy be placed in one protection domain or should the object be split so it can be placed in different domains? Languages such as C++ and Smalltalk place objects in contiguous memory. If protection hardware is not fine-grained enough to distinguish between portions of an object, it is not possible to place portions of a single object in different protection domains.

3.3.1.1 Parent More Trusted than Child

Consider first the case of a two-level implementation hierarchy where the parent is more trusted than the child. That is, the parent's protection domain includes all of the access rights of the child's protection domain and more. Figure 3.1 shows the different possibilities. The boxes are protection domains and the circles are objects. S_p is the parent subject, S_c is the child subject, O_p is the portion of the object belonging to the parent, and O_c is the portion of the object belonging to the child.

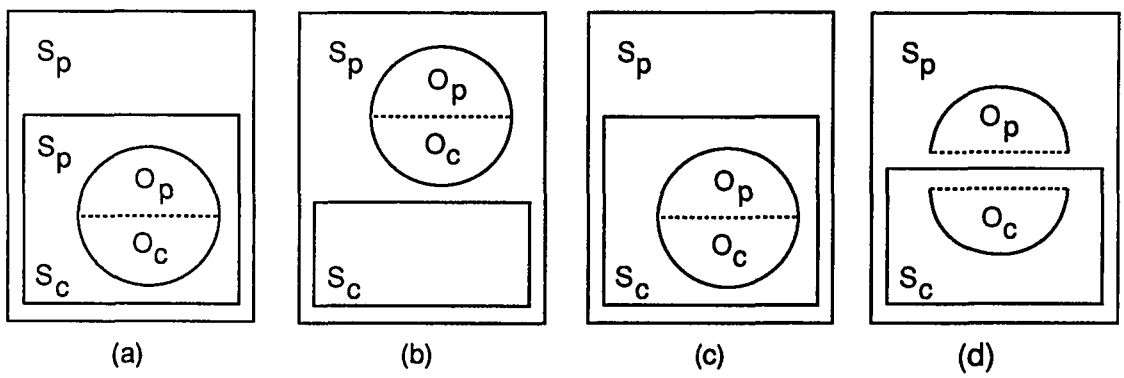

Figure 3.1: Unified or Split Objects

Assume that an entire object, both parent and child portions, must be in contiguous memory and that an object can only be protected by hardware in a single protection domain. In which protection domain should the object then be placed?

The first possibility is to have the parent use only the child's protection domain and then place the object in that domain. Figure 3.1a shows the parent subject having the ability to use either the more powerful domain or the more restrictive domain. Using the more restrictive

domain would prevent the parent from causing any side effects that the child could not cause itself. This can be easily implemented by essentially making a copy of the code for the parent subject and giving the copy the same protection domain as the less trusted child subject. This is appealing for the cases in which it may be applied because no protection boundaries need to be crossed, which improves performance. However, it is too restrictive for the general case because the parent might need to cause side effects that are only possible in the more powerful protection domain; for example, objects representing operating system services such as **Semaphore** objects will need to be able to cause side effects on other operating system objects such as **Process** objects which represent processes.

The second possibility is to leave the parent in the more powerful protection domain and place the entire object in that domain as in figure 3.1b. Clearly, this is not appropriate because the child would be prevented from manipulating its own portion of the object. The child in an implementation hierarchy defines its own portion of an object and the parent has no knowledge of the structure of the child's portion so the child must be able to modify that portion.

Therefore, if the object must be in a single protection domain and the parent needs to be able to be in a more powerful protection domain, the object should be placed in the child's protection domain where it is accessible by both the parent and the child as in figure 3.1c. In that situation the parent cannot trust that the child has left the parent's portion of the object intact because the child has complete access to the object. This is acceptable only if one of the following special cases is true:

1. The parent has not defined any data in the object. If there is no data, there is nothing for the child to modify. Few situations are covered by this case because classes usually define data; however, it is easy for the compiler or run-time system to verify whether the parent has defined data.

2. The parent causes no side effects based on the data in the object that the child does not have the right to cause itself. For example, the parent might only modify the object itself, not other objects or other protected data that the parent has the ability to modify, such as "class" data ("class" data is global information for a class). If a parent can cause side effects on protected data, the less trusted child can circumvent the protection by

manipulating the parent's portion of the object. This can cause the parent to modify the protected data to the child's advantage.

3. At the beginning of every method call, the parent validates the data that it has defined in the object to ensure that any side effects that the parent will cause will not violate trust. For example, perhaps some data items will cause correct operation if they are within a particular range of values. If that is so, the parent must check to make sure the data items are still within the correct range; it cannot trust that the data items have been left intact because the less trusted child may have modified them.

The latter two cases cannot be automatically enforced, so they require careful analysis by the programmer for each situation and are prone to error. In addition, even the last case does not guarantee correct operation in all situations: if the child cooperates with another concurrent thread of control, the other thread could modify the data during the execution of a parent method after the parent had verified the data.

The same analysis applies to a multi-level protection hierarchy at the points at which the trust levels change, not just a two-level hierarchy, The analysis can be applied iteratively.

Thus, placing the entire object in a single protection domain is not appropriate for the general case in an implementation hierarchy where child subjects are less trusted than their parent subjects.

3.3.1.2 Untrusted Parents and Mutual Distrust

Up to this point I have only been analyzing the case where a parent is more trusted than a child. Does it make a difference if the child in the implementation hierarchy does not trust its parent? This could be the case if either child and parent mutually distrust each other or if the child is more trusted than the parent[1].

Essentially, these variations make very little difference to the results of the analysis. If the parent is less trusted than the child the roles would simply be reversed in the analysis. If mutual distrust existed between the parent and the child, the possibilities are more restrictive

[1] Practically speaking, it is not very likely that a child will be more trusted than its parent; for example, an operating system is not likely to make a subclass of a class defined in an application. However, it is theoretically possible.

than a more-trusted/less-trusted relationship: a unified object would have to be made accessible to both the parent and child, and they would both have to protect against unauthorized modification by the other by verifying the data on every method call, which in general is not practical.

Thus the result of the analysis is the same no matter what the trust relationships are: objects should be able to be split across protection boundaries.

3.3.2 Objects Split Into Different Protection Domains

Since placing an object in a single protection domain is not appropriate in all cases when an implementation hierarchy is split into different protection domains, objects should be split into the different protection domains as in figure 3.1d. The portions of the object that belong to each level in the implementation should be placed in the protection domain of the subject they belong to. The child then delegates or forwards methods it does not recognize to its parent.

Additional problems are caused by splitting an object into different protection domains. Some inheritance-based languages allow child subjects to access directly the data members of parent objects. Clearly this cannot be allowed across protection boundaries. It is not usually allowed in delegation-based languages, and [Sny86] advocates that all access to parent instance variables should be through methods even when protection boundaries do not exist [Sny86].

Some languages also distinguish between methods that are callable only by child subjects and methods that are callable by all subjects, for example C++ with its **protected** and **public** methods. Unfortunately, this cannot be strictly enforced across protection boundaries because an extra child can be easily created that makes the protected methods available to other subjects. Language restrictions can still be used, but programmers of the more trusted subjects should not assume that enforcement of the restriction is strict.

Another question is whether true delegation should be used between the child object and the parent object as opposed to simple forwarding. If true delegation is used, a method call by the parent object could result in a call back to the child object if the child overloaded a parent method. Since the parent does not trust the child in this case, it may be that forwarding is better to prevent the parent from invoking an untrusted method when it is expecting a trusted method. Method calls to untrusted subjects have to be handled carefully, as discussed in section 3.2. Changing an inheritance hierarchy into a hierarchy that uses forwarding would

change the semantics that delegation would have preserved, so it may not always be preferable to use forwarding, but delegation across a protection boundary should not be used blindly to replace inheritance.

3.4 Summary

These are the general principles exposed by this analysis:

1. Untrusted subjects should be permitted to invoke methods only on objects to which they have obtained access rights.

2. Untrusted subjects should be permitted to invoke only selected methods on the objects to which they have access.

3. Objects passed in as parameters to a method call across protection boundaries should be validated to ensure that the caller has the right to use those objects. In addition, if the objects that are passed as parameters do not trust the caller, the objects should be validated to ensure they are of a type that the called method expects.

4. Untrusted subjects should be permitted to create only objects that belong to subjects that have chosen to allow it. Untrusted subjects should only be permitted to delete objects that they have created.

5. The object types that can be polymorphically replaced by an object of an untrusted subtype should be restricted to only selected types.

6. If an implementation hierarchy is split across protection boundaries, the portions of the objects belonging to the parent and child should be handled in one of two ways:

   (a) If the parent does not need to cause side effects in its own protection domain, the parent should instead restrict itself to the child's protection domain so that the complete objects can be placed in that domain.

   (b) If the parent does need to cause side effects in its own protection domain, the objects should be split so that the parent's portions are in the parent's protection domain and the child's portions are in the child's protection domain. Method calls that the child does not implement should be delegated or preferably forwarded to the parent.

Chapter 4

The Choices Implementation

This chapter is a high-level description of the implementation of object-oriented hierarchies across protection boundaries in the *Choices* operating system. This implementation has been designed using the principles discussed in chapter 3. Details of the implementation are in chapter 7.

The purpose of this implementation is to develop a working model of a mechanism that supports object-oriented hierarchies across protection boundaries. The model should be useful as a vehicle to study the general problems of splitting object-oriented hierarches and as an example to be imitated in other object-oriented languages and systems. The goals of this implementation are as follows:

1. To support the strict enforcement of protection. An untrusted user must not be able to circumvent the mechanism to abuse protected resources. This is not to say that an implementor of a protected resource cannot inadvertently allow protection violations by careless implementation of methods, but the mechanism itself must not be able to be compromised and it should assist the implementors of the protected resources whenever the assistance can be feasibly provided.

2. To provide language transparency whenever possible. Language transparency is necessary to provide a uniform object-oriented programming paradigm.

3. To perform adequately. The time taken to cross protection boundaries should be similar to non-object-oriented systems that provide protection.

4. To provide more than two protection levels. This is not of essential importance to the implementation, but it is included to improve the generality of the mechanism and to enable the structuring of the operating system into more-protected and less-protected portions.

The two major constraints on this implementation are the use of the C++ language and the use of traditional processor hardware. *Choices* is written in C++, and a transparent C++ interface across protection boundaries is desired. *Choices* currently runs on National NS32332 and Intel 80386 processors, so the only hardware protection available is the memory management units and, in the case of the Intel 80386, segmentation.

C++ does not make a distinction between the implementation hierarchy and the type hierarchy, which simplifies my implementation. On the other hand, the compile time type-checking of C++ complicates my implementation because no information about the types of objects is available at run time. To get around that deficiency, *Choices* keeps track of the type hierarchy with first-class classes [IL90, MCK91].

4.1 Protection Model

The protection model that I have chosen is a partitioned rings of protection model. Each successive outer ring is less trusted than the further-in rings. Furthermore, each ring can be partitioned into mutually distrustful regions. In order to make the interface as transparent to C++ as possible, address spaces are shared across rings although rings cannot access the addresses of further-in rings. Partitions within a single ring share the same range of virtual addresses, but represent different memory in different address spaces.

Figure 4.1 shows an example with four rings of protection. A partition of one ring is called a *ring-partition* (a ring-partition corresponds to a `Domain` in *Choices*[RJC88, RC89]). The number in each ring-partition is the ring number, and the letters represent the partition identifier. The last letter in each identifier distinguishes a ring-partition from other ring-partitions that belong to the same partition of the next inner ring. The other letters identify the ring-partition in the next inner ring. Ring-partitions 3aba and 3abb share the same range of virtual addresses and mutually distrust each other, and ring-partition 2ab is mapped into the address space of both of them and is more trusted than both of them.

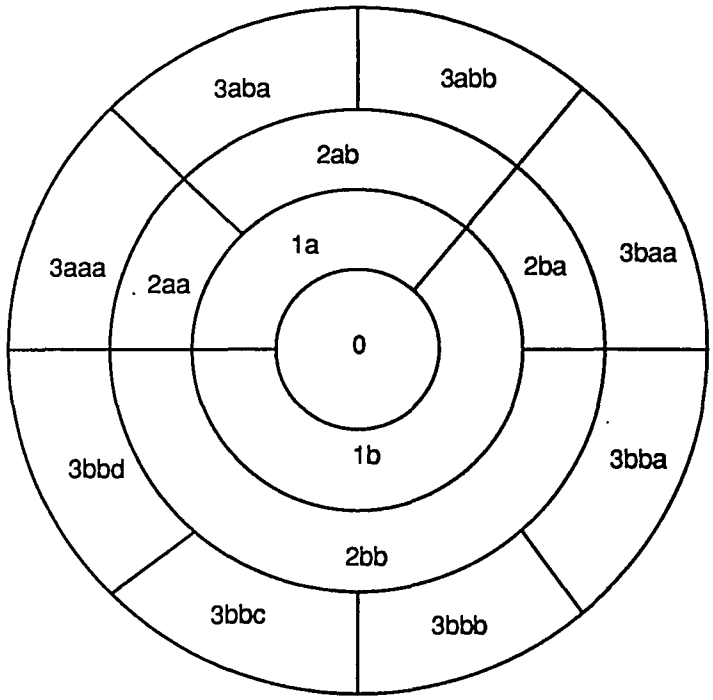

Figure 4.1: Four Partitioned Protection Rings Example

These are the reasons why this model of protection was chosen:

1. Due to the nature of the C++ language and our protection hardware, fine-grained protection is not practical. Instead, objects and subjects are grouped into relatively large portions that trust each other. Within a group of objects, no protection check overhead is incurred to invoke methods.

2. Providing intermediate protection rings allows efficient implementation of an object-oriented minimal kernel with shared services outside of the kernel. Layering the trust levels is efficient because it enables inner shared levels to use shared memory semantics rather than message passing semantics. That is, if the shared services are forced to be placed at the outer level in their own address spaces, messages have to be sent to them to gain access to their services. If they are at an inner trust level, they have direct access to the memory of the users they serve.

Additional intermediate protection rings could be useful to application developers who desire the ability to add their own protected services which are shared across several higher-level applications.

Both *incalls* from outer rings to inner rings and *outcalls* from inner rings to outer rings are supported. However, only incalls (the most common case) can take advantage of the shared address spaces. That is, code in an outer ring can pass references to data in its own ring-partition and expect the inner ring to be able to access it. Outcalls cannot pass references to data in their own spaces and expect the called ring-partition to be able to access the data because that would be a protection violation.

3. The partitioning of the outermost protection ring into separate address spaces that are mutually distrustful is common in operating systems. I chose to allow the intermediate rings to also be partitioned to enable a protected operating system research environment on a running system, and to allow higher-level shared application libraries to coexist on the same system without having to trust each other.

Even though any number of rings are allowed and even though any ring other than the innermost ring can be partitioned, I expect that the most useful configuration will be three rings where the intermediate ring contains shared operating system services that are outside of the kernel and only the application ring is partitioned, as shown in figure 4.2.

4.2 Hardware Protection Rings

The implementation of the partitioned rings of protection is processor dependent. The ideal hardware for the implementation of rings would have a protection level for every ring. Unfortunately, the processors on which *Choices* runs directly support only two protection levels. However, multiple rings can still be implemented by using some kinds of standard protection hardware. Two examples are presented in this section.

There are two major ways that most modern computer hardware provides protection: through paging and segmentation.

Paging systems provide virtual address spaces by means of page tables. Page tables contain page table entries. In addition to indicating where in physical memory a page resides,

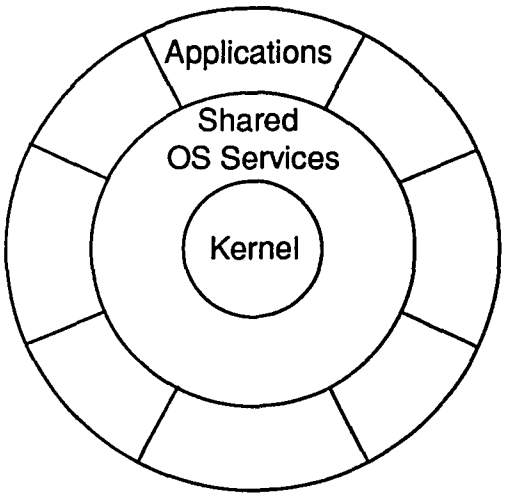

Figure 4.2: Basic Ring Configuration

a page table entry indicates protection on each page. Most modern Memory Management Units (MMU's) can indicate whether each page is writable or readable by two protection levels, supervisor and user.

Segmentation systems provide descriptors for each segment of memory. The descriptors indicate where a segment begins in physical memory, how long the segment is, and whether the segment is writable or readable by different protection levels.

The requirements of this design call for multiple hardware protection levels with shared address spaces. The implementation of proxies (described later) calls for portions of the kernel to be readable by all protection levels. Neither a paging system with two protection levels nor a segmentation system directly provides for these requirements.

4.2.1 Rings Using Segmentation and Paging

Processors such as the 80386 processor provide a combination of segmentation and paging. On these processors, creative use of segments can get around the problem of only two paging protection levels: the address space of the intermediate protection rings are placed above the outermost protection ring in the virtual address space, in reverse order from the outer protection rings through the inner protection rings. All protection rings outside of ring zero, the kernel ring, are run in user mode on the processor but have different segment descriptors associated

with each protection ring. All segments still start at virtual address zero so the kernel space can be shared, but the segments are limited in length, thus protecting more privileged data from the outer protection rings. Figure 4.3 shows an example address space layout for four rings on a system with a combination of segmentation and paging.

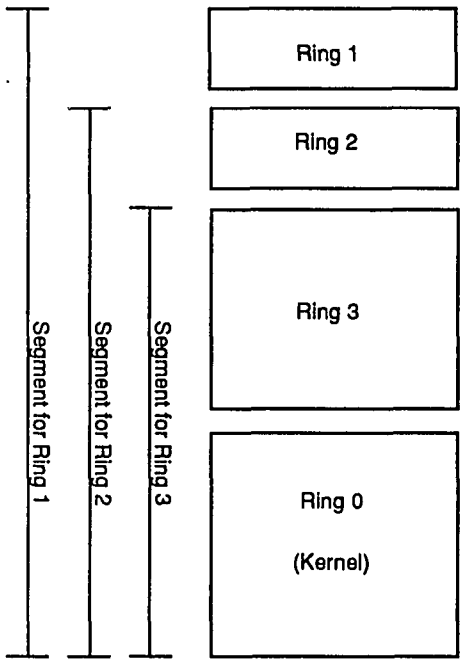

Figure 4.3: Example Address Space Layout on Segmentation/Paging System

4.2.2 Rings Using Two-Level Page Tables

Many processors do not provide segmentation but have a page table arrangement called two-level page tables. In a two-level page table system, the first level page table points to the second level page table and the second level page table points to the pages. The first level page table is much smaller than the second level page table, typically only one page.

For processors with no segmentation but with two-level page tables, multiple protection rings are achieved by having multiple first-level page tables per `Domain`. All rings outside of ring zero are run in user mode on the processors but have their own first-level page tables per `Domain`. For example, if there were four total rings, there would be three first-level page tables

per Domain. The first-level page tables all represent the same address space but have different address ranges accessible to user-mode code.

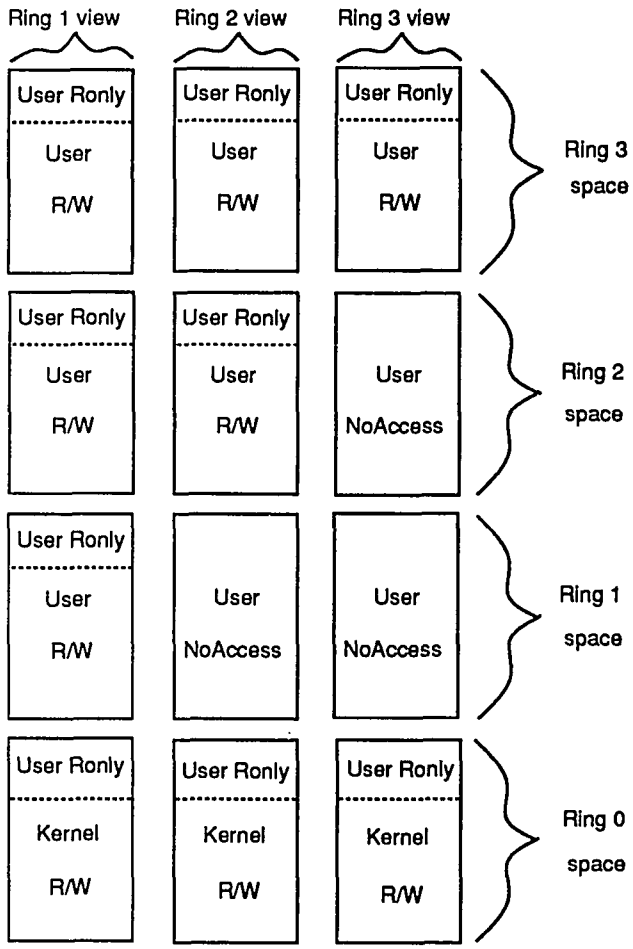

Figure 4.4: Example Two-level Page Table Address Space Views

Figure 4.4 shows an example of different views of the same address space on a system with two-level page tables and two hardware protection levels. The two protection levels are called "Kernel" and "User". The example shows four protection rings which requires three different views; ring zero can use any of the views because it operates in kernel mode. Each view corresponds to a first-level page table. Each first-level page table is divided into four areas corresponding to the four rings. Each ring has read/write access to its own space and all further-out rings. Each rings shows a portion that is read only by user because the implementation

of proxies requires a portion of address space in each ring that is readable by user mode but writable only by the kernel.

Loading a new page table at the time of the protection ring switch forces Translation Lookaside Buffer caches to be flushed. However, measurements show (see section 8.2) that doing so does not add much more time to a proxy call on the processors on which *Choices* runs. Nevertheless, there is some overhead, and hardware that can support multiple protection levels without needing to change page tables is better for implementing protection rings.

4.3 Proxy Objects

Objects are represented outside of their ring-partitions by *proxy* objects. A proxy gives a single ring-partition the *capability* to invoke methods on the object it represents. A method invocation on a proxy object is a *proxy call.* Proxies are instances of the ObjectProxy class. The user of an `ObjectProxy` is unaware of its existence and treats it as if it is the real object. The methods of ObjectProxy are stubs that are in a portion of the kernel that is readable to all applications. These stubs trap to kernel mode and translate the method calls into calls on the real object in the appropriate ring. The portion of the kernel that processes the traps is the *proxy trap handler.*

The real, or *proxied*, objects are instances of subclasses of the ProxiableObject class. The primary purpose of the ProxiableObject base class is to provide reference counting because C++ does not have garbage collection. ProxiableObject defines a reference count variable and methods to manipulate the count.

`ObjectProxies` are allocated from the user's ring, but in a section that is read-only and controlled by the kernel. `ObjectProxies` are automatically allocated (or stripped off) by the proxy trap handler whenever an instance of a subclass of ProxiableObject is returned from another proxy call or passed in as a parameter. If a proxy already exists from the same ring-partition to the same proxied object, the proxy is reused (it is looked up in a hash table).

Every application is started with a single proxy to an object called the `SystemInterface`; after that the application obtains all proxies through proxy calls. One call that the default application initialization code makes is to obtain a proxy for its `NameServer`. The `NameServer` simply maintains a table that maps strings to particular objects that are instances of subclasses

of ProxiableObject and allows applications to look up the objects by name. Since the lookup is a normal proxy call, a proxy is automatically allocated for the object when the call returns. This is the mechanism used to make objects available so any application can obtain proxies to them on demand.

In addition to being able to obtain proxies to pre-existing objects, a mechanism is provided to allow applications to create new objects by using the C++ **new** keyword on selected classes. Using that mechanism creates a protected object and returns a proxy for that object. This is discussed further in section 4.4.3.

4.4 Proxify++

To assist with addressing the concerns discussed in chapter 3, I have created a tool called *Proxify++*. This tool examines the definitions of C++ classes and generates information about each class. Proxify++ recognizes a new keyword called **proxiable** that identifies methods and constructors that can be used across protection boundaries. This keyword is stripped out by the C-preprocessor when the C++ compiler examines the class definitions, so no modifications to the C++ compiler are needed.

An external tool like Proxify++ is only necessary because the C++ language does not have enough information available at run-time and is not expressive enough to be able to be extended within the confines of the language. A language that was based on the principles of *reflection*, the ability to modify the computational model of a language [Mae87, Fer89], might not require an external tool. For example, a language like CLOS [Kee88] has many more reflective abilities than C++.

The **proxiable** keyword is a hint to the system that the programmer has designed privileged code which is safe to be used by untrusted code. The run-time system (proxy trap handler) prevents untrusted code from using privileged code that is not marked **proxiable**. A method that is marked with the **proxiable** keyword is known as a *proxiable method*, and a class that has at least one proxiable method is known as a *proxiable class*.

Proxify++ has four outputs:

1. Proxy table.

2. Alternate class description header files.

3. Constructor stubs.

4. Method stubs.

Each output is described more fully in the following subsections.

4.4.1 Proxy Table

Proxify++ generates a table of information about each proxiable method. The table, called the *proxy table*, lists the number of parameters and the types of parameters that are pointers, and the type of the return value. The methods are grouped by class.

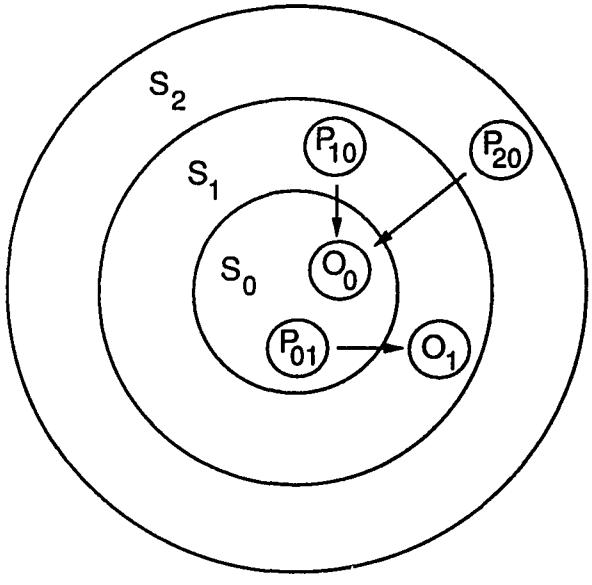

Figure 4.5: Three rings with proxies

The proxy trap handler uses this table to look at every parameter and return value to every proxy call. Parameters on incalls are treated very much like return values from outcalls, and return values from incalls are treated very much like parameters to outcalls. The trap handler uses the table to perform the following operations:

1. First, the proxy trap handler verifies that the caller has the right to make the proxy method call. This is the control mechanism mentioned in section 3.1.2. For example in

figure 4.5, if subject S_1 uses proxy P_{10} to invoke a method associated with object O_0, P_{10} is validated to make sure that S_1 can use it, and the method is verified to make sure it was one marked with the **proxiable** keyword.

2. Proxies are automatically allocated and stripped off, for both parameters and return values. For example, if S_1 makes an incall to a method of S_0 which returns a pointer to O_0, the proxy trap handler will automatically return proxy P_{10} to S_1. If S_1 later makes another call to a different method of S_0 and passes in P_{10} as a parameter, the proxy trap handler will automatically strip off the proxy so S_0 can directly access O_0.

   If S_1 repeatedly calls the method of S_0 which returns a pointer to O_0, the proxy trap handler will reuse the previously allocated P_{10} proxy rather than continually allocating new ones (the object is looked up in a hash table). This is important to prevent using up too many proxies on repetitive operations.

   If a third ring were involved, a proxy can be stripped off and another one allocated for the same parameter. For example, if P_{20} is passed as a parameter to an incall from S_2 to S_1, it will be stripped off and P_{10} will be allocated and passed in its place. This is done because proxies only map objects to one partition of one ring.

3. For all proxies that a subject passes in as parameters or returns, the proxy trap handler verifies that the ring-partition that the subject is in has the right to use that proxy. This implements the control mechanism of section 3.1.3. For example, if S_2 made a call to S_0 and attempted to pass in P_{10} then that call would be rejected because S_2 does not have the right to use P_{10}. (In normal operation S_2 would never attempt to do use P_{10}, but the programmer of S_2 could be maliciously trying to break security).

4. For proxies to objects that do not trust the caller, the type of the object is checked to verify that it is of the correct type or a subtype in the type hierarchy. The need for this was discussed in section 3.1.3. For example, if S_1 calls to S_0 and passes in P_{10} then O_0 is checked for type. If an object of type Process were expected and instead O_0 is of type Semaphore, that would be rejected. (Once again, compiler type checks prevent this from happening, but a malicious programmer could circumvent compiler checks).

5. The proxy trap handler controls at what point an untrusted subtype can take the place of a trusted parent type by only allowing this when the parent's class definition is marked with the **proxiable** keyword. The need for this was discussed in section 3.2. For example, say a method on S_0 expects a parameter of type Semaphore and S_1 passes in O_1 which is an instance of an untrusted subtype of Semaphore. The proxy trap handler would then allocate P_{01} for S_0 to use, but only if the Semaphore class definition has been marked with the **proxiable** keyword; if it has not been marked **proxiable** then the call is rejected. The reason for this is to only allow outcalls to this untrusted object if the Semaphore type has been specifically designed to allow outcalls.

6. If a parameter points to data that is not an instance of a subclass of ProxiableObject, the proxy trap handler verifies that the address of the data is in a ring that the caller has the right to access. The assumption here is that the amplified-privilege called method will read or write at that address and it ought to be an address that the caller may read and write. For example, if this check were not performed and S_1 called in to a method of S_0 which reads from a file into a buffer or writes into a file from a buffer, S_1 could easily read or write any value into memory in ring zero by passing in a pointer to memory in ring zero. This is a result of sharing addresses between rings.

   Only one level of pointer indirection is allowed for parameters to proxied methods; that is, Proxify++ does not allow "**" pointers. The reason for this is that it is impossible in C++ to know the format of the data and the checks cannot be performed; in a language that had parameters more completely specified this limitation would not be necessary. For the same reason, pointers to structures that contain pointers must not be passed to a proxy call; Proxify++ does not enforce this, but it is up to the programmer of the trusted method to enforce it.

4.4.2 Alternate Header Files

Another output of Proxify++ is alternate class description header files. These are the class descriptions of proxied classes that applications include. Only methods and constructors that are marked **proxiable** are copied to these header files; all other details are removed. This allows compile-time checks to notify application programmers if they accidentally try to call

methods that are not proxiable or try to use any member variables which are not present in the proxy. Since all implementation details are eliminated, the alternate header files have the added advantage of preventing recompilation of code that uses a proxied class if the interface does not change.

Alternate header files would of course not be needed for languages such as CLOS which do not require class descriptions to be exposed to users of the classes.

4.4.3 Constructor Stubs and Deletion

The third output of Proxify++ is constructor stubs. If a constructor is marked as **proxiable**, then an object of that type may be created by another ring. This is the control mechanism on creation of objects discussed in section 3.1.4. Proxify++ recognizes **proxiable** constructors specially and prepares a separate list of these constructors. This list is used to create constructor stubs to be linked with applications. This enables application programmers to do a **new** on a proxied class as if it were local, and to get back a proxy to the newly created object.

Proxied objects cannot be directly deleted by users of proxies. When the user of a proxy calls the destructor through **delete**, the proxy is deleted instead. When the proxy is deleted, the reference count on the proxied object is decremented by one. (When the proxy was first allocated, the reference count on the proxied object was incremented by one because the proxy referenced the object). When the reference count on the proxied object becomes zero because it is not being used by any other object, the proxied object is then deleted. This ensures that an application cannot delete an object that it did not create or that is still in use.

4.4.4 Method Stubs

The fourth output of Proxify++ is method stubs. Proxify++ generates these method stubs for every proxiable method and they are linked with applications. The purpose of method stubs is to be inherited by subclasses of proxiable classes; they are not used by calls directly to proxied objects. The method stubs implement the splitting of objects in an implementation hierarchy across protection boundaries that was discussed in section 3.3.2.

Since C++ allocates objects out of contiguous memory, separate C++ objects are used for the portions of a class hierarchy on different sides of a protection boundary. The child portion of the object inherits method stubs which invoke the appropriate methods on the parent object

through a pointer to a proxy to the parent object that is saved in the child object. Space for this pointer is reserved in every `ProxiableObject`, and the pointer is filled in when the constructor stub of the proxiable parent class is called by the constructor of the subclass. The implementation of ProxiableObject is copied into every application so that is why there are data variables in ProxiableObject available for the child portion of the object; this is in contrast to non-copied proxied classes which are viewed by applications to have no data variables because the applications use the alternate header files which have the implementation details removed. Section 5.4 gives more details on how this inheriting of method stubs work.

Note that this scheme only forwards methods to the parent objects, it is not true delegation. That is, once the method has been forwarded to the parent object, and if the parent calls another method in the same class that the child has overloaded, the child's method will not be invoked. In order to simulate true delegation, a pointer to the original object has to be passed as a parameter to the parent method. This is a recognized way to simulate delegation in C++ [JZ91].

Dynamic determination of parents as is provided in delegation-based languages is not currently implemented, but it could be provided through the use of a method on ProxiableObject that allows an application to set the pointer that is used by the inherited method stubs.

4.5 Hierarchy of Type Hierarchies

Every partition of every ring can independently extend the type hierarchy. An extension of the type hierarchy in one partition of a ring is not seen by other partitions of that ring. In order to keep track of these independent extensions of the type hierarchy, an unusual data structure has been put in place that represents this hierarchy of type hierarchies. This hierarchy has a type hierarchy for each ring-partition in the ring-partition hierarchy.

Figure 4.6 shows an example of an extension to the type hierarchy in one application. Each class in the type hierarchy is represented by an object that keeps track of types; those objects are shown in the figure. Classes that are at points in the type hierarchy that can be extended have an object representing them in every partition of each further-out ring; in the figure, ProxiableObject, Semaphore, and BinarySemaphore are points in the type hierarchy that are exposed outside of the kernel to allow applications to add subtypes. These duplicates maintain

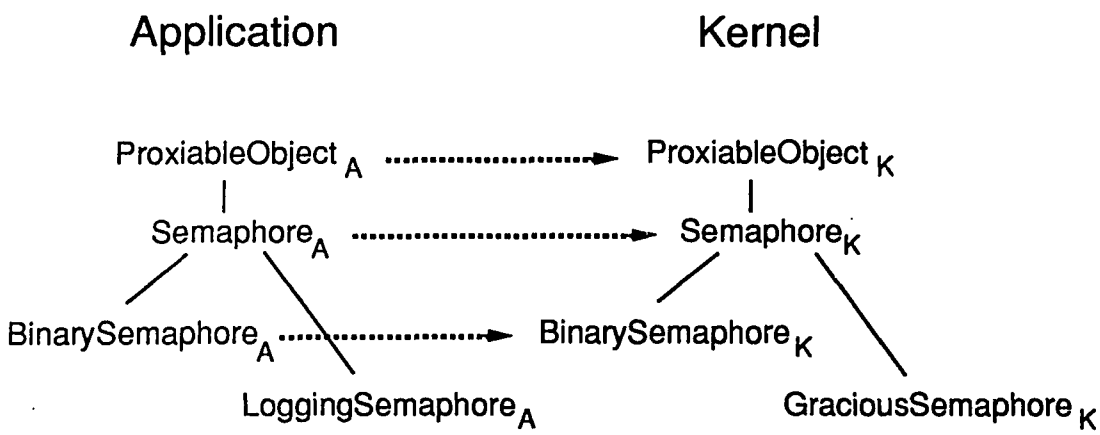

Figure 4.6: Example Hierarchy of Hierarchies.

a pointer to their originals as represented by the dotted lines in the figure. GraciousSemaphore is a type that is not exposed outside of the kernel, and LoggingSemaphore is type that is defined by an application and unknown to the kernel.

The operation that checks types follows the dotted-line pointers only in the direction of the arrow. In the figure, $\text{GraciousSemaphore}_K$ is considered a subtype of Semaphore_A but $\text{LoggingSemaphore}_A$ is not considered a subtype of Semaphore_K. In this way, each application can have its own view of the type hierarchy.

4.6 Implementation Limitations

I have already pointed out some differences between a normal C++ program that has one large unprotected address space and my implementation which splits the program across protection boundaries. Here are some additional limitations that I have placed on the implementation. These are not fundamental limitations, but I chose them for simplicity:

1. No direct function calls are supported across protection boundaries, only method calls. Traditional operating systems provide only function calls, but since I had the method call proxy mechanism in place there is no longer any reason to support ordinary function calls. Any miscellaneous function calls can be made to be methods on the SystemInterface class.

2. Multiple inheritance across protection boundaries is not supported. *Choices* does not use multiple inheritance and it would introduce significant complexity.

3. Calls between two unrelated **Domains**, that is, *crosscalls*, are not supported. It would require a major change to the *Choices* process subsystem to support that. I think that such calls should be implemented like remote procedure calls which switch the processor context to a server process rather than staying in the context of one process.

4. Outcalls are only supported to the **Domain** that a process was originally created in and to **Domains** that could have been reached through incalls when the process was created. For example, if a process that started in a particular application **Domain** calls a kernel method, that method cannot call out to a method in a different application **Domain**. That would have all the same problems as a crosscall.

4.7 Summary

This implementation satisfies the stated goals: it supports the strict enforcement of protection, provides language transparency, performs adequately, and provides for more than two protection levels. The implementation has been useful as a model to study general problems of object-oriented hierarchies across protection boundaries.

Chapter 5

Examples

This chapter provides examples of using the proxy subsystem. The examples demonstrate concepts that I have described in earlier chapters.

5.1 Printing a Simple Message

As every student of Computer Science knows, the first example that should be given for any new system should show how to print a simple message.

```
#include "OutputStream.h"
int
main( int, char ** )
{
    StandardOutput->formattedWrite( "hello sailor!\n" );
    return( 0 );
}
```

Figure 5.1: Printing a simple message.

Figure 5.1 shows how to print a simple message using the proxy subsystem. StandardOutput is initialized by application start-up code and points to an instance of BufferedOutputStream. BufferedOutputStream is defined in application space and is a subclass of OutputStream (a subclass of ProxiableObject) which is both defined in the kernel and copied into application

space. StandardOutput is of type "OutputStream *" so assigning a `BufferedOutputStream` to it takes advantage of polymorphism.

The formattedWrite() method (which is similar to printf() in C) is inherited from OutputStream. FormattedWrite() calls write() to write out the formatted data. Write() is defined by BufferedOutputStream to collect data in a local buffer and to call write() on a second `OutputStream` when a newline is reached.

This second `OutputStream` is in the kernel, and the `BufferedOutputStream` keeps a proxy to that `OutputStream`. Thus a call to write() on the proxy will trap to the kernel and invoke the proxied method there. A pointer to the buffer in the application is passed through, and the write() method in the kernel directly accesses the buffer. The proxy trap handler recognizes that the pointer parameter is not a subclass of ProxiableObject and verifies that the pointer refers to an address in the application's ring.

Note that because the OutputStream class is both in the kernel and copied into the application, the definition of the class is used by the application in this scenario for both a locally defined object (the `BufferedOutputStream`) and a proxy. Non-virtual functions on the proxy will not trap to the kernel where the proxied object is but instead will execute locally. This succeeds in this case because there are no member variables in the OutputStream class for those non-virtual functions to work on; the proxy would not contain the member variables if there were any. Also, all virtual functions in this class are proxiable so the virtual function numbers always match between the application and the kernel.

5.2 Pointers to ProxiableObjects

Figure 5.2 shows an example proxy call that both passes in a pointer to a `ProxiableObject` and returns a pointer to a `ProxiableObject`.

```
FreeRunningTimerStar SystemTimer = StandardNameServer->
                lookup( "SystemTimer", FreeRunningTimerClass );
```

Figure 5.2: Example of passing and returning ProxiableObjects.

StandardNameServer is a variable that is initialized for all applications and points to a proxy to the `NameServer` for that application. FreeRunningTimerClass is also initialized for all

applications and points to a proxy to an instance of the kernel class called Class. (The Class class is used to keep track of the class hierarchy at run-time, that is, first-class classes.) This proxy call to the NameServer::lookup() method traps to the proxy trap handler which recognizes the two parameters. The first parameter is a pointer to something other than a subclass of ProxiableObject; the proxy trap handler range-checks the pointer to ensure that this points to the application's space. The second parameter is a pointer to a proxy; the proxy trap handler validates the proxy to ensure that it is a valid proxy for the application, and then strips off the proxy so lookup() can directly access the `Class` object.

Lookup() looks up the object that had been given the name "SystemTimer" and returns a pointer to the object. The proxy trap handler takes that pointer and allocates a new proxy for that object and returns a pointer to the proxy instead of the real object.

The "Star" variable that is assigned the return value in this example is used for reference counting. If the SystemTimer variable is deleted or set to zero, the proxy will automatically get unreferenced so it can be deallocated.

5.3 Creating a Proxied Object

Applications may create proxied objects that are of a class that has a constructor marked **proxiable**. For example, figure 5.3 shows a subset of the header file describing the Semaphore class in the kernel.

```
#include "ProxiableObject.h"
class Semaphore : public ProxiableObject {
protected:
    int count;
    int maxCount;
public:
    proxiable Semaphore( int initialCount, int maxCount = 65535 );
    proxiable ~Semaphore();
    proxiable virtual void P();
    proxiable virtual void V();
};
```

Figure 5.3: Semaphore class description.

Proxify++ examines that header file and generates an alternate header file that looks the same except that the **protected** member variables are removed. If there were any non-proxiable methods then those would also be removed.

Proxify++ also generates a constructor stub so an application can create a **Semaphore** in the kernel and obtain a proxy to it. For example, figure 5.4 shows a code fragment that creates a

```
Semaphore * sem = new Semaphore( 1 );
sem->P();
delete sem;
```

Figure 5.4: Creation and deletion of a Semaphore.

proxied **Semaphore** object in the kernel, makes a call to the P() method, and deletes the proxy that is returned. Deleting the proxy deletes the object in the kernel. If the application neglects to delete the proxy, the proxy will be deleted when the application exits.

In addition, it is possible to use the syntax of declaring memory for a proxied object instead of creating it with **new**. Figure 5.5 shows an example of that. This allocates memory on the

```
{
    Semaphore sem( 1 );
    sem.P();
}
```

Figure 5.5: Declaration of a Semaphore.

stack and then calls the constructor stub for Semaphore. The constructor stub traps to the kernel to create the **Semaphore** and gets the proxy. After the trap returns, the stub saves a pointer to that proxy in a variable called "_real" which is defined by the **ProxiableObject** class.

Then when P() is called, a method stub that was also generated by Proxify++ is called. That method stub is shown in figure 5.6.

The stub simply forwards the call to the proxy. Note that this is less efficient than if the object had been created by a **new**, but the syntax and semantics of a locally declared object are preserved.

At the end of the program block the compiler automatically calls the Semaphore destructor, which deletes the proxy, which deletes the **Semaphore** object in the kernel.

```
void Semaphore::P() {
    return ((Semaphore *) (ProxiableObject *) _real)->P();
}
```

Figure 5.6: Semaphore::P() method stub.

A proxied object can also be declared at global scope. If that is done, the constructors of the object and the proxy are called when the application is initialized and the destructors are called when the application exits.

5.4 Inheriting Across Protection Boundaries

Applications may also create subclasses of proxiable classes. Figure 5.7 shows an example. This

```
#include "Semaphore.h"
class LoggingSemaphore : public Semaphore {
protected:
    int V_count;
public:
    LoggingSemaphore( int initialCount );
    virtual void V();
};

...

void LoggingSemaphore::V() {
    V_count++;
    Semaphore::V();
}
```

Figure 5.7: Example subclass across protection boundary.

subclass counts the number of calls to V() and inherits P(). The portion of an instance of class LoggingSemaphore that belongs to the subclass, namely the V_count member variable, will be in application space, and the portion that belongs to Semaphore will be in the kernel. This requires that the parent has a **proxiable** constructor.

A new on LoggingSemaphore will call the constructor of LoggingSemaphore. The constructor of LoggingSemaphore automatically calls the constructor of Semaphore, which in this case is a

stub that traps to the kernel to create the kernel portion of the instance. When the constructor stub returns, it fills in the "_real" pointer defined in the application portion of the instance (in ProxiableObject).

Calls to the parent methods go through the method stubs described in the previous section. Methods that are not overloaded inherit the stubs, and the stubs forward calls through the proxy to the parent object. The explicit call to Semaphore::V() inside of LoggingSemaphore::V() also uses the stub.

The implementation of ProxiableObject is copied into every application, so that is why the "_real" variable is available in the application object. Also, that means that the portion of the object in the application (the LoggingSemaphore portion as opposed to the Semaphore portion which is in the kernel) has its own reference count, and that method calls to methods of ProxiableObject apply to the local portion of the object and do not make proxy calls. Only calls to the non-overloaded method stub for Semaphore::P() result in proxy calls.

Note also that the implementation of the portion of the **LoggingSemaphore** object in the application is defined in C++ to have the Semaphore class as a parent class, but that the definition of Semaphore that the application sees from the alternate header file has no data items, only methods. Therefore no space is reserved in the application portion of the object for the Semaphore class, even though space is reserved for the copied ProxiableObject class.

5.5 Dynamic Determination of Parent

This is not currently implemented, but a method could easily be provided to allow dynamic determination of the proxy parent. An example of this is shown in figure 5.8.

In the figure, the parent of the **LoggingSemaphore** is dynamically set to be a different **Semaphore**. The original parent **Semaphore** and proxy that are created as a side effect of creating the **LoggingSemaphore** are automatically deleted because "_real" is a "Star" variable; a "Star" variable automatically unreferences the object that it was originally set to when it is re-set.

This technique could also be used to create an instance of a subclass of a proxied class that is not allowed to be created by an application; that is, a class that does not have a **proxiable** constructor. For example, suppose that the Semaphore class did not have a **proxiable**

```
void
ProxiableObject::setParent( ProxiableObject * newParent )
{
    _real = newParent;
}

...

    Semaphore * sem1 = new Semaphore( 1 );
    Semaphore * sem2 = new LoggingSemaphore( 0 );
    sem2->setParent( sem1 );
```

Figure 5.8: Dynamically Determining Parent Object.

constructor but an application could get a proxy to one through its **NameServer**. Then when an application creates a new **LoggingSemaphore** instance, no **Semaphore** object and no proxy would be created because there will be no constructor stub to call. Instead, a local object would be created with only method stubs for the Semaphore methods and a null "_real" pointer. However, the created object could still be useful because the setParent() method could be used to point the parent of this object to a pre-existing **Semaphore** object.

5.6 Outcalls

An outcall is a method call from an inner ring to a further-out ring. The inner ring obtains a proxy to an object in an outer ring through the passing of a parameter that points to the object. The inner ring can then use that proxy to make the outcall. Figure 5.9 shows an example of making an outcall.

The kernel defines the interface to which it will call out. By marking the FaultHandler class definition with the **proxiable** keyword, the kernel indicates that it is willing to accept subclasses from outer rings when it expects parameters of type FaultHandler. When the application calls setFaultHandler(), a proxy to the applications **MyFaultHandler** is automatically allocated for the kernel. SetFaultHandler() saves the proxy in the **Process** object. Later, when a page fault occurs, the page fault handler retrieves the proxy from the **Process** and makes the outcall to the application's fault() method.

```
From kernel header files:

proxiable class FaultHandler : public ProxiableObject {
...
    proxiable virtual void fault( char * address,
                                  AccessType attemptedAccess );
};
...
class Process : public ProxiableObject {
...
    proxiable virtual void setFaultHandler( FaultHandler * );
    proxiable virtual FaultHandler * faultHandler();
};


From an application:

class MyFaultHandler : public FaultHandler {
...
    virtual void fault( char * address,
                                  AccessType attemptedAccess );
};

...
    thisProcess()->setFaultHandler( new MyFaultHandler() );


From kernel page fault handler:

    thisProcess()->faultHandler()->fault( addr, access );
```

Figure 5.9: Example outcall.

5.7 Simulating Delegation

If a parent method in the kernel wants to be able to call a support method which may be overloaded by a child outside of the kernel, a pointer to the child object must be explicitly passed in to the parent method. Figures 5.10 and 5.11 shows examples of this kind of parent and child.

In this example a call to V() on a LoggingSemaphore will invoke doV() in the kernel, passing in a pointer to the child object. Note that the Semaphore class has to be marked **proxiable** for

```
proxiable class Semaphore : public ProxiableObject {
...
    proxiable virtual void V();
    proxiable virtual void doV( Semaphore * me );
    proxiable virtual void supportV();
};

...

void Semaphore::V() {
    doV( this );
}

void Semaphore::doV( Semaphore * me ) {
    me->supportV();
    ....
}

void Semaphore::supportV() {
    /* don't do anything, just for overloading by child */
}
```

Figure 5.10: Example parent class for delegating.

```
class LoggingSemaphore : public Semaphore {
...
    int V_count;
...
    virtual void V();
    virtual void supportV();
};

...

void LoggingSemaphore::V() {
    doV( this );
}

void LoggingSemaphore::supportV() {
    V_count++;
}
```

Figure 5.11: Example child class for delegating.

the kernel to accept a subclass of the Semaphore class from outside the kernel. Then doV() uses that pointer to outcall to the child's supportV() method. This style of simulating delegation in C++ is discussed in more detail in [JZ91].

5.8 Multi-ring Hierarchies

All of the examples I have given in this chapter up to this point have only dealt with two rings, kernel and application. Hierarchies can also span more than two rings. For example, a parent class can be defined in the kernel, a child class defined in an intermediate ring, and a child of that class defined in an application.

The example *Choices* application called 'timings' makes use of such a multi-ring hierarchy. That application tests and times all possible combinations of incalls and outcalls between three rings. It makes use of a TimingInterface class which is defined in the kernel, a child SubTiming-Interface class which is defined in the 'load1' intermediate ring example, and a child of that class called SubOutCallInterface which is defined in the 'timings' application. The concepts are similar to the examples which have already been shown so additional figures will not be provided at this point.

Chapter 6

Experience Moving the Filesystem out of the Kernel

In order to exercise the proxy subsystem, I did some work to enable the filesystem to operate outside of the kernel. The endeavor was successful, although there were several obstacles to overcome along the way. This chapter describes experience I gained by moving the filesystem out of the kernel to an intermediate ring.

Running the filesystem in an intermediate ring is a very good example of the use of rings in the proxy subsystem. This enables the kernel to be protected from filesystem code while still allowing the filesystem to take advantage of kernel services. In addition, the filesystem can still directly access memory of its client applications just as it could when it was inside the kernel.

6.1 Kernel-Specific Pieces Inside Filesystem

The first problem I ran into was that the filesystem was using PhysicalMemoryChain, a class that could not be moved out of the kernel. `PhysicalMemoryChains` are like pointers that refer to physical memory rather than virtual memory. Since it is undesirable to allow rings outside of the kernel to have access to physical memory, I removed all knowledge of `PhysicalMemoryChains` from the filesystem.

The filesystem used `PhysicalMemoryChains` to enable the virtual memory subsystem to use the filesystem to read into a page of physical memory before the page was mapped into a virtual address space. I removed that requirement by changing the virtual memory subsystem

to pre-map the physical page into a virtual address and then changing the filesystem to use only virtual addresses.

6.2 Splitting the Class Hierarchy

The next step was to decide where to split the class hierarchy between the kernel and the filesystem ring. Figure 6.1 shows a portion of the original class hierarchy and figure 6.2 shows the split class hierarchy. For more complete information on the *Choices* filesystem class hierarchy see [Mad92, MLRC88, MCRL89, WBJ90].

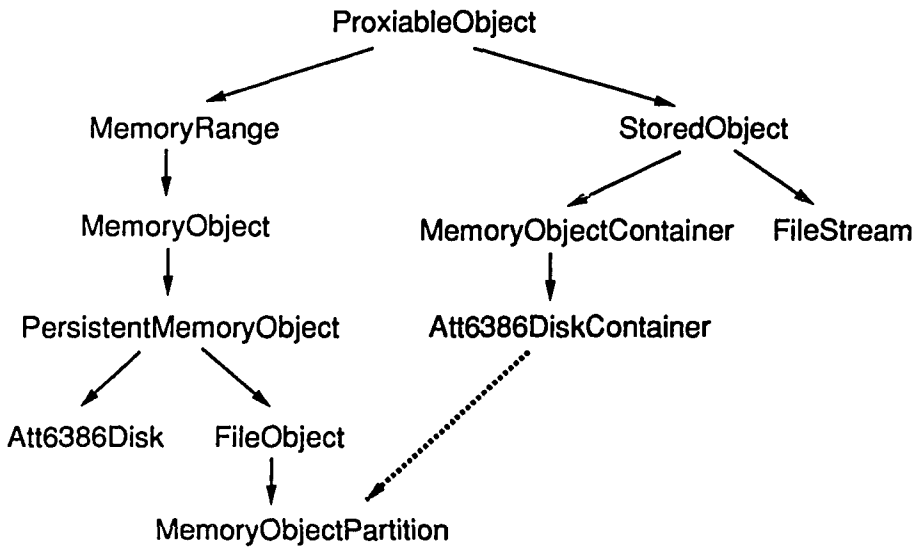

Figure 6.1: Portion of Original Class Hierarchy

In figure 6.2, classes without boxes in the normal font are defined in the kernel only, classes with boxes around them are defined in the filesystem ring only, and classes in bold font are defined in both rings. Solid lines with arrows indicate parent-child inheritance relationships. The dotted lines with arrows will be explained in section 6.2.3.

6.2.1 Filesystem Objects Split

The base class of much of the filesystem is **PersistentMemoryObject**. The parent of that class is MemoryObject which must remain only in the kernel because it is also the base class of the

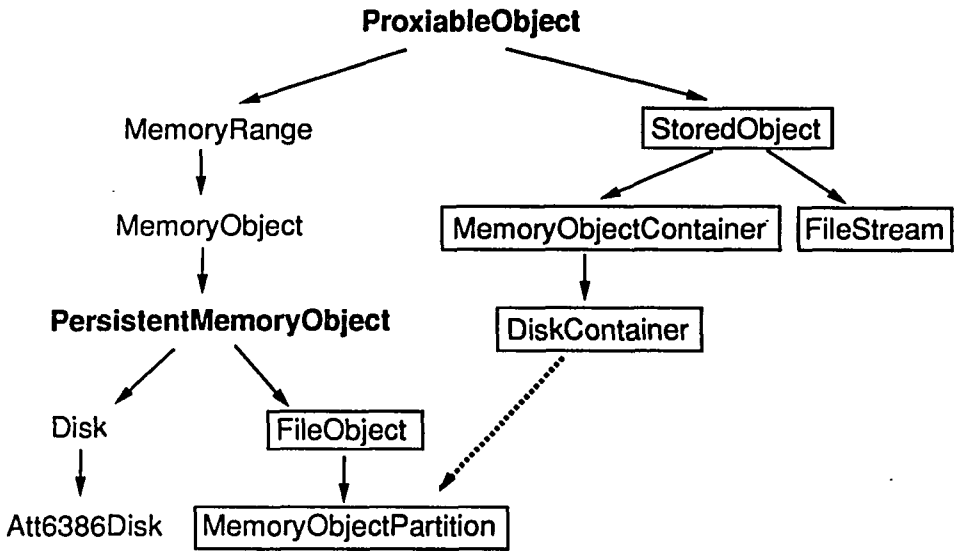

Figure 6.2: Split Class Hierarchy

virtual memory subsystem and it uses `PhysicalMemoryChains`. When an instance of any class is created in the kernel, the portion of the object belonging to that class and the portions belonging to all its parent classes are together in the kernel, so the MemoryRange class which is the parent of MemoryObject also remains only in the kernel.

The classes from PersistentMemoryObject and below are outside of the kernel in the filesystem ring (PersistentMemoryObject is in both rings, that will be explained in section 6.2.2). This means that every instance of a subclass of PersistentMemoryObject that is created in the filesystem ring is split into two parts, one part in the filesystem ring and one part in the kernel.

Unfortunately, MemoryRange contains two very basic member variables, the unit size and the number of units; together these contain the total size of the `MemoryRange`. Whenever a subclass of PersistentMemoryObject has a need to access those member variables, it must make a proxy call to the kernel where the portion of the object that contains those variables exists.

There is no fundamental reason why the kernel needs to have control over a portion of these filesystem objects, it is only necessary because of the use of `PhysicalMemoryChains` by MemoryObject. It would be better if there were a redesign so that either MemoryObject no longer uses `PhysicalMemoryChains` or so that PersistentMemoryObject no longer has MemoryObject as a parent. A major reason why PersistentMemoryObject was designed to be a subclass of Mem-

oryObject in the first place was so files could be easily made to map into virtual memory; that could instead be accomplished by the creation of a special subclass of MemoryObject specifically for that purpose that represents a file, and then changing the parent of PersistentMemoryObject so that most of the filesystem is not under MemoryObject.

The point is that the best design of an object-oriented hierarchy in the absence of protection boundaries may not be the best design when protection boundaries are introduced.

6.2.2 A Copied Class

In the original class hierarchy, PersistentMemoryObject was not only the parent of many filesystem classes, it was also the parent of Att6386Disk. Att6386Disk is the class that implements the disk driver on the AT&T 6386 PC version of *Choices* (corresponding classes in other versions of *Choices* are also subclasses of PersistentMemoryObject). The disk driver needs to remain in the kernel, so its parent PersistentMemoryObject also needs to be in the kernel.

One possibility would be to keep PersistentMemoryObject only in the kernel and to proxy it like MemoryObject and MemoryRange. PersistentMemoryObject also has some basic member variables so this would have the same problem that MemoryRange has: child classes need to make proxy calls to access those variables. There is an additional problem in PersistentMemoryObject because "inheritance" across protection boundaries in the proxy subsystem is implemented as forwarding instead of true delegation. The PersistentMemoryObject::asA() method calls supports() which is expected to be overloaded by child classes, and if PersistentMemoryObject were only in the kernel then asA() would not be able to find the child's supports() method. It would have been feasible to change the interface to simulate delegation like the example in section 5.7, but there is a better solution.

Instead of leaving PersistentMemoryObject only in the kernel, I chose to copy it so it is also in the filesystem ring. When an object that is an instance of a subclass of PersistentMemoryObject is created in the filesystem ring, the PersistentMemoryObject methods execute there in the filesystem ring. On the other hand, when the filesystem ring needs to access a subclass of PersistentMemoryObject that is in the kernel, such as the disk driver, the methods of PersistentMemoryObject are accessed through a proxy and execute in the kernel. Note that the user of the proxy in this case does not have the benefit of using an alternate header file for

the definition of the PersistentMemoryObject class; the same header file is used for both local objects and proxies. This has the following implications:

1. All **virtual** methods of the copied class must be **proxiable.** If there are any virtual methods that are not proxiable, the method numbers will not match and methods called on the proxy will not be the correct methods.

2. Non-virtual methods of the copied class, if any, will execute locally even on a proxy. This is not a problem if the method does not attempt to access any local member variables, but most methods do. In general there should be no non-virtual methods.

3. The compiler does not prevent attempted accesses to member variables on the proxy even though those variables are not there. **Public** member variables, if any, are a problem. **Private** member variables, which are only accessible by the class itself, are not a problem if there are no non-virtual methods. **Protected** member variables, which are allowed to be accessed by child classes, are also not a problem (assuming no non-virtual methods on the copied class) because child classes are children of the copied class and not of a proxy and thus these member variables will be available locally.

These limitations are a problem in C++ but I think they will not not be a problem in other object-oriented languages that have more information about class interfaces available at run time; that is, a run-time system that can tell the difference between an instance of the copied class and a proxy. However, in C++ it is best to avoid copying classes if practical.

6.2.3 Rearranging Part of the Hierarchy

In the original class hierarchy as shown in figure 6.1, the disk driver is made up of not only Att6386Disk but also Att6386DiskContainer. The dotted line in the figure is there to indicate that Att6386DiskContainer creates a MemoryObjectPartition. Since the machine-dependent Att6386DiskContainer would have to remain in the kernel if I had left the hierarchy as it was, I would have also had to leave StoredObject, MemoryObjectContainer, FileObject, and MemoryObjectPartition in the kernel as well.

Instead of leaving all those classes in the kernel, I found a way to rearrange part of the hierarchy to avoid it. I isolated the machine-dependent portion of Att6386DiskContainer in a method in

Att6386Disk and made a machine-independent DiskContainer class to call that method. To prevent the machine-independent DiskContainer class from having to have knowledge of a machine-dependent class, I inserted a machine-independent Disk class above Att6386Disk (the Disk class was first described in [Kou91]). After DiskContainer calls the method on Disk that is overloaded by Att6386Disk to return the machine-dependent information, DiskContainer creates the necessary `MemoryObjectPartition` as indicated by the dotted line in figure 6.2.

Thus by only slightly changing the design of the hierarchy I was able to move four more classes out of the kernel. I expect that a similar technique can be used by other people attempting to introduce protection boundaries into class hierarchies.

6.3 Minimal Filesystem Remains in the Kernel

Although I was successful in getting the filesystem to run outside of the kernel, a version of the filesystem code still remains in the kernel. Although this filesystem currently shares much of the same source code as the one that runs in the intermediate ring, it should be stressed that the two filesystem incarnations are independent of each other. The only thing that they share is the disk driver. For example, even though the filesystem in the kernel happens to include a StoredObject class, in figure 6.2 I only show it in the intermediate ring because the intermediate ring filesystem has no interaction with or knowledge of the StoredObject class in the kernel.

These are the reasons that a filesystem still remains in the kernel:

1. The paging system requires a non-paged filesystem. When the paging system needs to read a page from the disk or write a page to the disk, the code and data that implements the disk reads and writes must be locked in memory and not be paged out itself. Currently only the kernel is guaranteed to be locked in memory.

   In addition, a complicated filesystem is currently required in the kernel because the paging system on the Multimax version of *Choices* uses one of the main partitions on the disk for its backing store along with other files. If the paging system used a separate disk partition for backing store, a much simpler filesystem could be used.

2. The function that loads and runs programs off of the disk is still in the kernel. It would be possible for the function to be modified to make an out-call to the intermediate ring

filesystem, but it would take some redesign including having the kernel allocate buffers in the intermediate ring so the outcall can access the buffers. A better possibility is to move the function that loads and runs programs out to the intermediate ring.

3. The kernel filesystem is currently used to load and run the program for the intermediate ring. This requirement could be avoided if the booters/launchers were able to load both the kernel and the intermediate ring program at the same time.

Chapter 7

Implementation Details

This chapter gives implementation details on the proxy subsystem of *Choices*. It is intended to be useful to those who desire to thoroughly understand how the subsystem works and to those who desire to modify it.

Note: all filenames used in this chapter are relative to the top of the Stable.9.23.1991 *Choices* source node. The notation <Machine> indicates the machine name (Att6386, Multimax) and <Processor> indicates the processor name (i386, NS32332). Appendix A contains a summary of the source files and appendix B contains a summary of the output files.

7.1 Proxify++

Proxify++ is a major component of the *Choices* proxy subsystem. By examining *Choices* header files, Proxify++ automatically generates new files that are compiled and linked with the *Choices* kernel and with applications. This section describes the Proxify++ tool in detail.

7.1.1 Usage

Proxify++ is normally only invoked from the *Choices* makefiles with a single set of options. See section 7.2 for a more complete description of the makefiles.

Proxify++ accepts options followed by pathnames of files to process. The options are as follows:

-all - process all files included from the specified file. This is an option that the makefiles currently use; the makefiles generate a source file that '#include's all files that are to be proxified and passes that file to Proxify++. Without this option, only the single specified file is processed.

It is faster to process all the header files at once than to do each one individually. Also, '-all' makes it possible to process only the interesting subset of the *Choices* header files rather than all the *Choices* header files. The alternate header files that Proxify++ generates follow the '#include' hierarchy of the original source files. Without '-all', every header file in the entire *Choices* system must be individually run through Proxify++ to ensure that every header file that is included from the interesting subset gets an alternate header file generated for it. With '-all', Proxify++ generates alternate header files for the interesting subset and all files they include so the rest of the *Choices* header files can be ignored.

If the '-all' option is on, proxy tables and alternate header files are generated for every included file no matter how deep the level of include nesting. Method stubs and constructor stubs, on the other hand, are only generated for files directly included by the specified file.

-tableonly - only generate proxy tables and not the other three outputs. This is currently not used but is necessary for use on some header files when '-all' is not used.

-version - include a call to the REQUIREDVERSION() macro in every alternate header file generated. This is used from the current makefiles to implement per-class version checking (see section 7.9).

-h*host* - run the 'proxify' executable on a different host. This is used on the Multimaxes so 'cpp' runs locally where it has access to Multimax /usr/include source files and so the 'proxify' executable runs on a Sun-4 machine. The g++ compiler cannot correctly compile 'proxify' on the Multimax, and the Sun-4 machines are significantly faster than Multimaxes.

cpp options - all remaining options are passed to the C-pre-processor. Most useful options are -I, -D, and -U.

7.1.2 Output Files

Proxify++ generates four kinds of files. These files are moved around by the makefiles, so users generally are not exposed to the original output file names; for convenience, both the original name and the name that the makefiles currently rename the files to are listed here. The filenames referred to in this subsection are relative to the directory that Proxify++ is run in; currently Proxify++ is run in the Configure/System/<Machine> and Applications/LoadableExample/-<Machine> directories.

The names of the files that Proxify++ generates have the prefix of the source file that is being processed. The example prefix I will use here is 'ProxyIncludes', the file prefix currently used in the makefiles.

These are the four files that Proxify++ generates:

'ProxyIncludesT.cc' - proxy table file. When the '-all' option is off, this file goes in the subdirectory 'T'. The makefiles rename this to "PROXYTABLES.cc". This file contains tables describing every method marked with the keyword **proxiable** in the source header file. If the '-all' option is on, tables also are generated for all files the source header file includes no matter how deep the include nesting level. The structures that are filled in are defined in Includes/Common/ProxyTable.h.

Proxify++ knows that at the top of the object hierarchy there are classes called Object and ProxiableObject, and it only generates proxy information for these classes and for subclasses of ProxiableObject. If the names of these base classes are ever changed, simply change the initialization of the variables "subBaseClassName" and "baseClassName" respectively in proxify.cc.

ProxyCallAssist() uses the proxy tables when a proxy call occurs. The tables contain per-class information and per-method information. The per-class **ProxyTable** structure contains the following elements:

char * tableClass - the name of the class this table belongs to.

char isProxiable - a bit that indicates whether the class definition itself is marked with the **proxiable** keyword. This is used to determine whether a method parameter of this type will accept a subclass of this type that is defined outside of the protection ring of the method. For more details, see section 7.6.1.

int numMethods - the number of **proxiable** methods in this class.

MethodTableEntry ** methods - a pointer to the per-method information.

int numConstructors - the number of **proxiable** constructors in this class.

MethodTableEntry ** constructors - a pointer to the per-constructor information.

The per-method `MethodTableEntry` structure contains the following elements:

char * returnClass - the name of the return type of this method, if the modifier of the return type is a supported type.

char returnModifier - the modifier of the return type. The only supported modifiers are '*', '&', and "Ref". Ref is a suffix on the class name and the others are normal language modifiers for pointers. Ref is indicated in this table by an 'R' character and '*' and '&' are represented by themselves.

int realMethodNumber - the method number of the method in the proxied class. Since not all **virtual** methods are **proxiable**, the method numbers that the user of a proxied class sees are likely to be different than the method number on the real class; this element provides for the translation.

int numParams - the number of parameters for this method.

char ** paramClasses - the types of pointer parameters. If a parameter is not a pointer, the entry for that parameter is null. A parameter is a pointer if it has a single '*' or '&' modifier; Stars, Refs, and multiple '*'s or '&'s are not supported for parameters on **proxiable** methods.

header file(s) - alternate header file(s). The alternate header files go into the 'H' subdirectory and are the same name as the original. If the '-all' option is on, all header files directly and indirectly included from the source file as indicated by the '#line' or '# N' directives in the output from the C-pre-processor have alternate header files generated for them; if '-all' is off, only a header file for the given source is generated. The makefiles move the alternate header files out of the 'H' subdirectory to the $(TOP)/Includes/$(MODULENAME)/-$(MACHINE) directory (for an example see MODULENAME = System and MACHINE = Att6386).

The alternate header files contain the class definitions that are in the source files but with only the **proxiable** methods left in. Data definitions and non-**proxiable** methods are removed. Other things that are left in are "#include", "extern", "enum", "typedef", and "const" statments. If "class", "struct", and "union" statements have no **proxiable** methods in them, they are simply declared to exist but their contents are not copied.

'ProxyIncludesC.cc' - constructor stub file. If the '-all' option is not on, this file goes in the subdirectory 'C'. The makefiles rename this file to "CONSTRUCTORS.cc". There is an empty constructor stub in this file for each **proxiable** constructor. If the '-all' option is on, constructor stubs are generated for each file directly included in the first level file; if the '-all' option is off, constructor stubs are generated for only the first level file itself. This file is not used directly, but it is compiled to determine the long C++ name for the constructors.

'ProxyIncludesS.cc' - method stub file. If the '-all' option is not on, this file goes in the subdirectory 'S'. The makefiles rename this file to "$(MODULENAME)STUBS.cc". There is a method stub in this file for each **proxiable** method. If the '-all' option is on, method stubs are generated for each file directly included in the first level file; if the '-all' option is off, method stubs are generated for only the first level file itself. These stubs are linked with applications. Each stub simply invokes the method by the same name as itself using the "_real" member variable. The stubs are not used for normal proxy calls, they are only used when applications allocate their own space for a proxy instead of doing a **new** to create them, and when applications create subclasses that "inherit" from a proxied class. For more details see section 7.3.2.

7.1.3 Source Files

The source for Proxify++ is in the Tools/Proxify++ directory. Proxify++ itself is a shell script that runs the C-pre-processor, does a little more filtering, and runs the 'proxify' executable. The extra filtering that it does is to remove lines that begin with the keywords **operator** and **asm** (possibly preceded by whitespace) because 'proxify' does not understand those.

The 'proxify' executable is made up of parts from several different sources. The 'parser.y' file is based on the public domain YACC grammar for C++ by Jim Roskind (jar@ileaf.com).

The 'lexer.l' file is derived from a lexer by Tony Sanders (sanders@sanders.austin.ibm.com). Those were taken by Andrew Grimshaw (grimshaw@cs.virginia.edu) and Ed Loyot (ecl2v@-virginia.edu) of the University of Virginia to make a compiler for a language called MPL that is a superset of C++. The front end of that compiler has the ability to generate a parse tree for the entire language and to print the source code from the parse tree. I took that portion, slightly modified some of it and fixed some bugs, and added the file 'proxify.cc' to process the parse tree. Pieces that were added to the program to support Proxify++ have "#ifdef PROXIFY" around them so if that define is turned off, it should revert to printing out the source code in its entirety.

This is a short description of each file in the Tools/Proxify++ directory:

Proxify++.sh - Source for the Proxify++ shell script.

common.h - A header file included in all the source files. Contains general definitions.

declarations.cc - Contains functions used to declare identifiers.

error.cc - Contains functions for error handling.

lexer.l - Contains rules for FLEX to lexically analyze source files and feed them to the parser.

main.cc - Contains the main() function which drives the different phases.

parser.y - Contains the YACC grammar and actions to build the parse tree.

print.cc - Contains the methods that print the nodes of the parse tree. The classes that these methods are a part of are defined in "syntax_tree.h".

proxify.cc - Contains most of the portions of the program that are unique to Proxify++. In addition to initialization and support functions, it contains proxify() methods on parse tree nodes corresponding to the print() methods in the file "print.cc".

symbol_table.cc - Contains methods that implement the symbol table.

symbol_table.h - Contains class definitions for the symbol table.

syntax_tree.cc - Contains methods for the classes that define the nodes of the parse tree, except for the print() and proxify() methods, and the type processing methods.

syntax_tree.h - Contains the definitions for the nodes of the parse tree.

tokens.h - Defines extra tokens not directly used by the parser.

types.cc - Contains functions and methods for type processing. Classes for these functions are also defined in "syntax_tree.h".

7.1.4 Debugging

Here are some hints for debugging Proxify++. The first thing to do is to make a simple source file and keep taking things out until you have narrowed it down to the piece that Proxify++ is having trouble with. If you ran Proxify++ from the makefiles and had a problem, you can begin from the preprocessed source that Proxify++ will have left in your Configure directory.

After you have got it down to the essential piece, take the following steps:

1. Log in to a Sparcstation, either from a console or through rlogin.
2. Go to Configure/Tools/Proxify++/sparc in your node.
3. Put your small test file there, for example 't.cc'.
4. Make sure that the test file compiles with g++. The error handling of Proxify++ is worse than with g++ so this is necessary to ensure a syntactically correct source file before giving it to Proxify++.
5. Run 'proxify' on it. Proxify is the executable that Proxify++ (a shell script) calls after running cpp; you can skip cpp and go straight to proxify. By default proxify generates four files and they are put in the four subdirectories C, H, S, and T. proxify requires a parameter for the prefix to use for those output files and then it reads from standard input, so you can use 'proxify t <t.cc' and it will make files with the prefix 't' in those subdirectories.
6. You can debug using gdb. Another helpful thing is to set the 'YYDEBUG' environment variable. Meaningful values are:

   2 - print only symbols found and proxify phase debugging messages. This is useful when the problem is in the proxify phase.

3,4 - also print yacc state reduction debugging messages. I have not found this to be very useful.

5 or higher - instead of state reductions, print out parse trees in a tree format during parsing. This is extremely useful when the problem is in the parse phase.

In order for all the debugging to work this way, Proxify++ should be compiled with all the debugging options on. The options are -DDEBUG_LEXER, -DDEBUG_PARSER, -DDEBUG_PRINT, -DDEBUG_PROXIFY, and -DYYDEBUG.

7.2 Proxify Makefiles

Makefiles perform an important role in the proxify process. The makefile for each load module that needs to have Proxify++ run for it must include Configure/System/ProxifyCommon.mk. That common makefile takes care of running Proxify++ and it takes care of compiling files to be linked with both the load module and with applications that need to make proxy calls into the load module.

To avoid unnecessary recompilations, ProxifyCommon.mk often keeps two copies of files that it generates. The copy is usually prefixed with "New" or "NEW". After ProxifyCommon.mk generates a "New" file, it compares the generated file with the previous non-New file to see whether anything has changed, and only updates the non-New file if they are different. Dependencies are based on the modification times of the non-New files, so recompilation is not done if the newly generated files are the same as the old ones.

7.2.1 ProxifyCommon.mk Inputs

ProxifyCommon.mk expects the following make variables to be set:

MODULENAME - The load module name. In Stable.9.23.1991 this is "System" for the kernel that is made in Configure/System/<Machine> or "Example" for the example loadable module that is made in Configure/Applications/LoadableExample/<Machine>.

PROXYINCLUDES - A list of the names of all header files that are to be proxified and have all four outputs of Proxify++ generated for them: proxy table, alternate header file,

method stubs, and constructor stubs. Each of the header files in this list becomes a "#include" statement in ProxyIncludes.cc.

ALLINCLUDES - A list of header files for which alternate header files should be kept if one is generated. Proxify++ generates alternate header files for all header files listed in $(PROXYINCLUDES) and all files that they include and all that they include, and so on. Only those listed in $(ALLINCLUDES), however, are saved in Includes/$(MODULENAME)/$(MACHINE). Each of the header files in this becomes a "#include" statement in AllIncludes.cc.

TABLEONLYPROXYINCLUDES - A list of the names of header files that are to only have the proxy table output of Proxify++ generated. Examples of such header files are those that are define classes that are compiled for the kernel and all applications such as Object.h and ProxiableObject.h. Generation of only proxy tables is implemented in the makefile by only indirectly including these files into ProxyIncludes.cc; each of the files in this list turns into a "#include" statement in TableOnlyProxyIncludes.cc which itself is "#include"d by ProxyIncludes.cc. With the '-all' option on Proxify++ turned on, this prevents Proxify++ from generating the method stubs and constructor stubs for these files. Also, by not including the files from $(TABLEONLYPROXYINCLUDES) in the $(ALLINCLUDES) list, the alternate header files that Proxify++ generates for these are not kept.

STARSOURCES - This is the list of "Star" files. These files are compiled once per machine type and loaded into the lib$(MODULENAME).a library in the current directory so they can be used by applications in addition to being linked in with the load module.

TOP - A relative pathname to the top of the source node. This defaults to "../../.." in Configure/ChoicesCommon.mk and is redefined by other common makefiles to "../../../.." when directory nesting is four levels deep.

MACHINE - The machine name, for example 'Att6386' or 'Multimax'. Defined in Configure/<Machine>Common.mk.

PROCESSOR - The processor name, for example 'i386' or 'NS32332'. Defined in Configure/<Machine>Common.mk.

STANDARDHVPATH - Standard include directories that are included for both the kernel and applications. This is set in Configure/ChoicesCommon.mk. This variable is used to compile some files for use in the application library lib$(MODULENAME).a.

COMMONHVPATH - This is only used if $(MODULENAME) is not "System". These are more include directories to use when compiling files for the library. This is set in Configure/Applications/ApplicationsCommon.mk.

PROXIFY - Path name to Proxify++. Defined in Configure/<Machine>Common.mk.

C++ - Path name to the C++ compiler. Defined in Configure/<Machine>Common.mk.

CONSTCOLLECT - Path name to a program that can collect the long symbol names used by the C++ compiler for constructor names. Source for this program is in Tools/Misc/-Constcollect.sh. That program requires the "NM" environment variable to be set with the pathname of the 'nm' program that goes along with the compiler. Defined in Configure/-<Machine>Common.mk.

CPPFLAGS - Flags to pass to the C++ compiler, including "-I" flags which point to include directories. These flags are passed to Proxify++ so it can find all the include files. Defined in Configure/<Machine>Common.mk.

COPTS - Options passed to the C++ compiler when compiling files for the application library. Defined in Configure/<Machine>Common.mk.

MV - Path name for the mv command. Defined in Configure/<Machine>Common.mk.

RM - Path name for the rm command. Defined in Configure/<Machine>Common.mk.

7.2.2 ProxifyCommon.mk Outputs

A makefile that uses ProxifyCommon.mk should include all files in the $(PROXIFIEDSOURCES) variable in the load module that it builds. Also, it should make lib$(MODULENAME).a by default in addition to its load module.

See appendix B for details about output files that ProxifyCommon.mk creates.

7.3 Control Flows

This section covers the flow of control of the proxy subsystem of *Choices* during run-time operation.

7.3.1 Control Flow of Method Calls

Assume that an application already has a valid `ObjectProxy` (section 7.3.5 covers how an application gets its first `ObjectProxy`). For example, every application has a proxy to its `NameServer`; a pointer to that `ObjectProxy` is in the StandardNameServer variable. In this section I will follow an example call to the lookup() method on the NameServer class. There are two lookup() methods in that class with different parameter types; I will follow the one that takes a pointer to a `Class` as the second parameter.

The class definition that the compiler uses to make the method call is not from the original "NameServer.h" that is in the kernel, it is from the alternate header file that is generated by Proxify++. In this case the compiler generates a call to virtual method number 19. The compiler looks up the 19th method in the virtual function table of the ObjectProxy class. The virtual function table for the ObjectProxy class is in the kernel but it is in a region of memory that is readable by all applications; this virtual function table is generated by the compiler from Kernel/ObjectProxyVtable.cc which is loaded into the application-readable region. The call then proceeds to method number 19 of ObjectProxy which is in ProcessorDependent/-<Processor>/ObjectProxyStubs.s. The stubs in that file are also in the application-readable section of the kernel.

The stub function traps to the kernel trap handler in ProcessorDependent/<Processor>/-<Processor>ContextSwitching.s and passes method number 19 in along with the original parameters to the method. The trap causes a switch to the kernel stack for the current process and a switch to kernel mode. The trap handler reserves space on the kernel stack for `ProxyReturnFlags` and for copied parameters before calling the ProxyCallAssist() function in Kernel/ObjectProxyKernel.cc.

ProxyCallAssist() does the following primary operations for this call:

1. validates the this pointer that was passed in to the method call. This validation verifies that the this pointer refers to a valid `ObjectProxy` for the `Domain` of the caller, and it is

necessary to ensure that the caller has the right to call methods on the proxied object. The validation is done by the isValidProxy() method on the ObjectProxyManager class in Includes/ObjectProxyManager.h.

For some compilers the this pointer may be the second parameter instead of the first; for a discussion see section 7.4.

2. determines the type of the call; that is, whether the call is an incall, an outcall, a crosscall, or a samecall. Only incalls and outcalls are supported, but the other kinds may be attempted and they need to be identified to give reasonable error messages. The example call to NameServer::lookup() is an incall.

3. obtains a reference to the `ProxyTableCopy` for the class of the proxied object. The `ProxyTableCopy` is a verified and slightly modified copy of the original `ProxyTable` that Proxify++ generated, in this case the NameServerProxyTable (for reasons for the copy, see section 7.3.4.6). A reference to the proper `ProxyTableCopy` is included in the `ObjectProxy` when the `ObjectProxy` is allocated.

4. verifies that the method number being called is valid for this class. There are 22 methods on the NameServer class, so method number 19 is valid.

5. obtains a reference to the `MethodTableEntryCopy` for the method. This is done by looking up the 19th element in the proxy table.

6. determines whether this method is a method that applies to the `ObjectProxy` itself rather than the proxied object. The four methods that apply to the `ObjectProxy` are those that are related to reference counting; see section 7.5.1 for more details. The lookup() method is not one of those special methods so ProxyCallAssist() continues.

7. looks up the real method in the virtual function table of the proxied object.

8. sets the this pointer for the real method to be a pointer to the real object.

9. verifies that the address for the return `Ref` is in a ring that is writable by the caller. lookup() returns a `ProxiableObjectRef`, which is a structure, not a simple value. The compiler allocates space for that return structure in the caller, not the callee, and passes

a pointer to that structure in on the call. The called method will write into that structure on its return. Since the called method will be running in the kernel ring and thus has the privilege to write anywhere in memory, ProxyCallAssist() here ensures that the caller does not pass in a pointer to memory that the caller does not have the privilege to write into. The ring of the address is determined by calling the lookupRing() method on the Kernel class which is defined in Kernel/Kernel.cc.

10. goes through each parameter, copying and translating where necessary. The `MethodTableEntryCopy` has the information about the number and types of each parameter. In my example, the first parameter is a "char *" and the second parameter is a "Class *". If any parameters were not pointers or were pointers that had a value of zero, they would just get copied to be ready for the call to the real method. Since these are both pointers and since this is an in-call, the function ProxyInCallParamAssist() is called for each parameter. For more details see section 7.6.1; in summary, the first parameter is range-checked to ensure it is in the caller's ring and the second parameter has the proxy to the `Class` object in the kernel stripped off.

11. sets up the `ProxyReturnFlags` for use by ProxyReturnAssist() after the method has returned. Values of variables that are needed at that time are stashed away in the `ProxyReturnFlags` which is in space on the stack that was reserved by the proxy trap handler.

12. returns to the proxy trap handler.

The proxy trap handler then calls the real method that was returned by ProxyCallAssist() as an element of the `ProxyReturnFlags`. When the method returns, control once again returns to the proxy trap handler. At that point the proxy trap handler calls ProxyReturnAssist(). Since NameServer::lookup() returns a `ProxiableObjectRef`, the pointer to the object was written out to the first word of the `Ref` structure; the proxy trap handler reads it out of there and passes it to ProxyReturnAssist().

The return values from an in-call need the exact same treatment as parameters passed in to an out-call. For that reason, ProxyReturnAssist() calls ProxyOutCallParamAssist() (covered in section 7.6.2) which allocates a new `ObjectProxy` for the returned object for the caller's `Domain`.

From there control returns to the proxy trap handler which writes the pointer to the `ObjectProxy` back out to the `Ref` and returns from kernel mode back to the calling stub. Finally, the calling stub returns back to the original caller.

7.3.2 Control Flow of Constructor Calls

This section follows the flow of control of a call to the constructor of an `ObjectProxy`. For this example I will follow a "**new** Semaphore" from within an application.

The first thing that is called is a stub defined in Libraries/SystemInterface/<Processor>/-Constructors.s. That file creates a definition for the macro "CONSTRUCT()" and then includes "CONSTRUCTORS.h". "CONSTRUCTORS.h" is a file generated by Proxify++ that is series of calls to the "CONSTRUCT()" macro, one for each constructor that was marked with the **proxiable** keyword. The parameters to the macro include the long C++ symbol name and a unique string for the constructor; for the example, those are "_9Semaphoreii" (for g++, different on cfront) and "SemaphoreConstructor0". The "CONSTRUCT()" macro uses the long symbol name as the symbol to identify the stub; that is the function that the compiler generates a call to when "**new** Semaphore" is used.

The stub calls the ConstructProxy() stub which is located in the application-readable section of the kernel in ProcessorDependent/<Processor>/ObjectProxyStubs.s, passing in the unique string identifier of the constructor along with the original parameters that the caller passed to **new**. ConstructProxy() traps to the kernel exactly like the other stubs in the same file, but with a method number of "-1".

Control flow then continues like normal **proxiable** method calls. ProxyCallAssist() detects the "-1" method number and calls ProxyConstructAssist() which is also in Kernel/Object-ProxyKernel.cc. ProxyConstructAssist() looks up the string "SemaphoreConstructor0" in the `NameServer` for the calling process, and retrieves a `ConstructorDescriptor`. The `ConstructorDescriptor` contains pointers to the real constructor, the `Domain` of the real constructor, and the `MethodTableEntryCopy` that describes the function; it is defined in Includes/-ConstructorDescriptor.h. ProxyConstructAssist() then returns that information to ProxyCallAssist() which continues like a normal method call.

The `ConstructorDescriptor` was installed into the `NameServer` by use of a `Constructor-Installer` as defined in Includes/Common/ConstructorInstaller.h. That file defines a "CON-

STRUCT()" macro that creates a **ConstructorInstaller** that is initialized with a static constructor. The makefiles generate a small file in Configure/System/<Machine>/CONSTRUCTORINSTALLERS.cc that includes the two files "ConstructorInstaller.h" and "CONSTRUCTORS.h" (which contains the invocations of the "CONSTRUCT()" macro). The static constructor of the **ConstructorInstaller** installs the **ConstructorInstaller** object into a linked list. Later, the initialization of the kernel calls ConstructorInstaller::install() to create the **ConstructorDescriptor** objects and install them in the **NameServer**.

After ProxyCallAssist() processes the parameters to the constructor like a normal proxied method call, it returns to the trap handler which calls the constructor. After the constructor finishes, it returns to the trap handler which then returns to user mode in the stub in ObjectProxyStubs.s. That stub then returns to the constructor stub in Constructors.s. Instead of returning directly to the invoker of **new**, however, the constructor stub calls _PostConstructProxy() in Libraries/SystemInterface/Portable/_Start.cc. That function does not do anything if the constructor was called with **new**, that is, if the original this pointer was zero (requires the use of the '-fthis-is-variable' option on g++). However, if the constructor was called because the user defined a subclass of "Semaphore" or because the space was preallocated by the user declaring a variable such as "Semaphore mySem" rather than doing a **new** on it, then _PostConstructProxy() saves a pointer to the just-created **ObjectProxy** in the "_real" member variable in the preallocated object. Later if any methods are called on that object, the method stubs that were automatically generated by Proxify++ in Configure/System/<Machine>/SystemSTUBS.cc use the "_real" variable to call the method on the proxy.

Before setting "_real", _PostConstructProxy() does some initialization on the object that would have been done by a constructor had the constructor been generated by the compiler. Namely, it calls the constructor for ProxiableObject and sets the vtable pointer to the class of the created object. This assumes that there are no other parent constructors that need to be called except for the ProxiableObject constructor (the ProxiableObject constructor calls the Object constructor), which is usually a valid assumption because class definitions in the alternate header files contain no member variables. It is possible for class definitions in alternate header files to inherit from classes other than ProxiableObject and other alternate class definitions, however, and if those classes have constructors then this implementation could be a problem because those constructors will not be called.

7.3.3 Calls to Non-Kernel rings

So far my examples have only shown incalls into the kernel. Incalls to rings other than the kernel and outcalls from the kernel take a couple detours from the control flow of incalls to the kernel.

For every proxy call initiated from rings other than the kernel, ProxyCallAssist() calls PROCESSOR_CONTEXT::saveUserStack() which is defined in Includes/ProcessorDependent/-<Processor>/<Processor>Context.h. On the i386 processor, this simply saves the value of the user stack pointer in the context object; on the NS32332 processor, this does nothing because there is a processor register that always contains the most recent value of the user stack pointer. All rings outside of the kernel are implemented using user mode on the processors, so the same stack pointer register is used for all those rings.

For proxy calls to rings other than the kernel, ProxyCallAssist() calls PROCESSOR_CONTEXT::setNewRing() which is defined in ProcessorDependent/<Processor>/<Processor>Context.cc. This method has two main purposes:

1. to switch the user stack. The first time that a process makes a call to a ring, setNewRing() allocates a stack for that ring for the process. The process cannot continue to use the user stack in the outermost ring because of a potential protection violation due to multiple processes: a different process executing in the outermost ring would be able to corrupt the stack of the process which is executing in a more privileged ring.

   SetNewRing() uses a portion of the kernel stack, the part that is used for copied arguments on calls to the kernel, to save state information about the call. The state information is saved in the form of a `UserProxyStack` structure which is defined in <Processor>Context.h. SetNewRing() saves the previous user stack pointer for the calling ring in the `UserProxyStack` and saves the current user stack pointer for the calling ring in the context where later proxy calls can find it. It also sets up a portion of the `UserProxyStack` to prepare for switching to user mode at the address of the proxied function in the called ring, and ensures that a return from the proxied function will trap back to the kernel.

2. to change the range of addresses that user mode can access. Each inner ring is allowed to access further-out rings but not further-in rings. On the i386 processor, this is accom-

plished by simply changing segments which were initialized by i386CPU::setRingRange in ProcessorDependent/i386/i386CPU.cc. On the NS32332 (or if "MULTIRINGTRANSLATION" is enabled on the i386), this is done by changing page tables. The TwoLevelPageTable `Translation` maintains separate top-level page tables for each ring outside of the kernel for every domain. The portions of that class in Kernel/TwoLevelPageTable.cc that implement this behavior are separated by "#ifdef MULTIRINGTRANSLATION". In addition, the <Machine>MMU::basicActivate() method in ProcessorDependent/<Processor>/<Processor>MMU.cc is given a pointer to the context of the process being activated so it can ask the `TwoLevelPageTable` for the page table of the currently active ring number.

Also, for proxy calls to rings other than the kernel, ProxyCallAssist() calls Process::setCurrentDomain to set the non-kernel `Domain` that the process is running in. This is used on subsequent proxy calls to determine the calling `Domain`.

When ProxyCallAssist() returns to the trap handler, the handler additionally saves the processor registers that the compiler assumes do not change (that is, non-volatile). If the called method only uses compiled code, that assumption will likely be true. However, the user-mode code could use assembly code or some other compiler that could modify the value of those registers and could be a privilege violation. After saving those registers, the proxy trap handler switches to user mode at the address of the proxied method.

The return address of the proxied method on the user-mode stack is set to be _ProxyReturnCall() which is in ObjectProxyStubs.s. This traps back to the kernel to the return assist trap handler in <Machine>ContextSwitching.s. That trap handler restores the compiler non-volatile registers and calls ProxyReturnAssist(). In addition to doing the things that it does on proxy calls to the kernel, ProxyReturnAssist() calls PROCESSOR_CONTEXT::restoreRing() in <Processor>Context.cc. This restores state from the `UserProxyStack` and restores the range of addresses that user mode can access. ProxyReturnAssist() also calls Process::setCurrentDomain() to restore the non-kernel `Domain` to the previous one.

7.3.4 Initialization of the Proxy System

This subsection covers the portions of kernel initialization that are related to the proxy system and protection rings.

7.3.4.1 APPLICATIONRING

The static constant APPLICATIONRING which is defined in Includes/MachineDependent/-<Machine>/<Machine>CPUConfiguration.h determines the number of rings defined in the system. The APPLICATIONRING is the protection ring of top level applications. Ring 0 is the kernel, and rings 1 through APPLICATIONRING-1 are the intermediate rings. If no intermediate rings are desired, APPLICATIONRING should be set to 1.

7.3.4.2 basicInitialize()

The <Machine>Kernel::basicInitialize() method sets several variables in the parent Kernel class which define the beginning and length of each ring. All intermediate rings are assumed to be of the same size and adjacent to each other in the memory map. See the comments at the beginning of MachineDependent/<Machine>/<Machine>Kernel.cc for more specific limitations.

7.3.4.3 _ringAddresses array

The Kernel::initialize() method in Kernel/Kernel.cc takes the variables defined by basicInitialize() and calculates the start and end of each ring and saves them in the Kernel::_ringAddresses array. It then invokes the <Processor>CPU::setRingRange() method to send the same information to the processor-dependent CPU class (used to initialize segments on the 80386 processor). After that, Kernel::initialize() builds a list of `VirtualMemoryRange` objects from the same information plus any other `PhysicalMemoryRange` objects that basicInitialize() may have passed back, and it passes the list to ADDRESS_TRANSLATION::init()[1].

7.3.4.4 Translation initialization

Both the Att6386 and Multimax ports use the TwoLevelPageTable class for their translation. That was discussed in section 7.3.3.

[1] Perhaps the _ringAddress array should be part of a separate object instead of being part of Kernel. Also, there is probably another way to get the ring information to the 80386 processor so the CPU would not have to get involved.

7.3.4.5 Kernel MemoryObjects

Later in Kernel::initialize(), `MemoryObjects` are allocated to represent each portion of virtual address space in the kernel. These `MemoryObjects` are then added to the kernel `Domain`. Figure 7.1 shows the layout of the memory that these memory objects represent in the kernel ring.

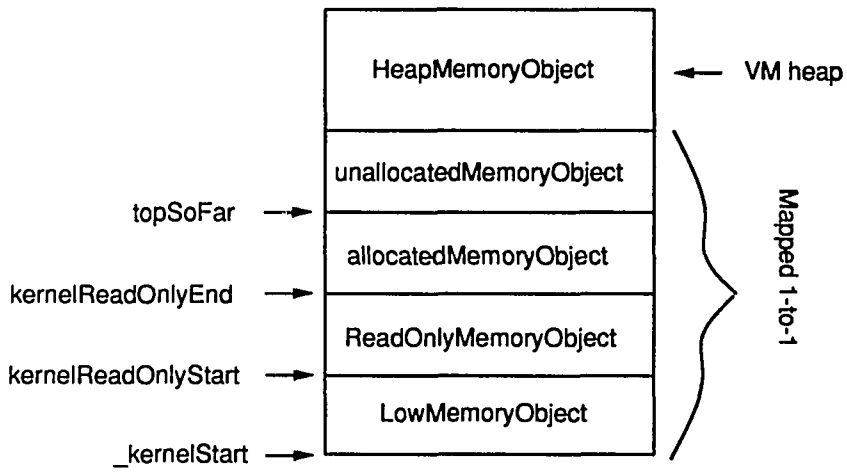

Figure 7.1: Kernel Ring Memory Map

The ReadOnlyMemoryObject covers the range of kernel text that is readable by all applications for use by the proxy mechanism. It is delimited by the variables KernelReadOnlyStart and KernelReadOnlyEnd. The LowMemoryObject covers kernel text below KernelReadOnlyStart. The allocatedMemoryObject covers all kernel text, data, and bss after that, as well as all memory allocated in the early initialization before virtual memory is enabled, up through the variable "topSoFar". After that is the unallocatedMemoryObject; this maps all remaining physical memory 1-to-1 into kernel virtual memory space so all physical memory can be easily accessed by the page fault handlers before pages are mapped into their ultimate virtual addresses. The unallocatedMemoryObject can be taken out when the virtual memory handler is changed to only read from and write into pages that are mapped in. The pages represented by the unallocatedMemoryObject are marked as not "preAllocated" by the PhysicallyResidentMemoryObject constructor so they can still be allocated by the heap manager.

After that is the HeapMemoryObject. This represents all the remaining virtual memory in the kernel. It is not mapped 1-to-1; it allocates extra virtual address space when needed. The HeapMemoryObject is not a `PhysicallyResidentMemoryObject` like the others; it is a `RootMemoryObject`. The source for both of those types of objects is in Memory/-DummyMemoryObject.cc

The figure does not show the machine-dependent areas that are also mapped 1-to-1. More `PhysicallyResidentMemoryObjects` are allocated for them and added to the kernel `Domain`. Currently there is only one allocated for the Multimax for the memory-mapped I/O area, and none on the Att6386.

7.3.4.6 ProxyTableInstaller

After VM is enabled, the first process continues at Kernel::main() which calls ProxyTableInstaller::install() in Common/ProxyTable.cc. That method goes through the list of `ProxyTables` that was created by the static constructors of the ProxyTableInstallers instantiated in PROXYTABLES.cc. For each `ProxyTable` in the list, the method creates a `ProxyTableCopy`, source code in Kernel/ProxyTableCopy.cc.

Each `ProxyTableCopy` is essentially the same as the corresponding `ProxyTable`. The only difference is that character string names of classes are replaced by pointers to `Class` objects after the string names are looked up in the `NameServer`. That is not the reason the copies exist, however; those transformations could have been done in place and in fact were in an earlier implementation. The reason the copies exist is that `ProxyTables` can be outside of the kernel, and the copies are all within the kernel. The copies are always easily accessible to the kernel whereas the originals may not be. More importantly, the copies are guaranteed by the kernel to be internally consistent; the kernel cannot trust a further-out ring to not modify the originals after they had been verified and the kernel would otherwise have to do many consistency checks on every proxy call into a further-out ring.

7.3.4.7 ConstructorInstaller

The next method that Kernel::main() calls is ConstructorInstaller::install(). See section 7.3.2 for information about initialization of constructors.

7.3.4.8 VersionInstaller

The version checking subsystem is not strictly part of the proxy subsystem but it is related. Kernel::main() initializes the version checking subsystem at this point. See section 7.9 for more information.

7.3.4.9 DefaultDomains

Kernel::main() also creates the default `Domain` objects for each ring from ring one trough APPLICATIONRING-1. The `Domain` for the kernel (ring zero) is created at the time the `Kernel` object was created because it is contained in the definition of the Kernel class. These `Domains` are arranged in a hierarchy of rings, and they are used as the default parents for the Kernel::buildProcessFromObjectFile() method. I provided default `Domains` because I expect that in most cases there will be only one `Domain` in each ring below APPLICATIONRING.

The default domains are saved in a global array called DefaultDomains. There is also a global array called DefaultDomainCleaners: this is used to prevent the cleaning up of the default domains until system shut down. Normally, the DomainCleaner class is used to clean up a `Domain` (in particular, to free all the proxies allocated for the `Domain`) when the last process in the `Domain` dies, but I want the default `Domains` to remain until the system is shut down. DomainCleaner is defined in Memory/Domain.cc and is simply an extra reference count for the cleaning up of the proxies in each `Domain` as opposed to the destruction of the `Domain` object. Keeping an extra reference to the default cleaners in DefaultDomainCleaners prevents the cleanup of the default `Domains` from occurring until the system is shut down.

7.3.4.10 Domain

The constructor of Domain does important initialization for the proxy system. Source code is in Memory/Domain.cc. The constructor creates one each of these for every `Domain`:

1. A `NameServer`.
2. A `MemoryObject` for the allocation of `ObjectProxies`. The Domain constructor makes this `MemoryObject` read-only for user mode and installs it in the `Domain`.
3. An `ObjectProxyManager`.

7.3.4.11 NameServer

The NameServers in Kernel/NameServer.cc maintain a mapping of strings to ProxiableObjects. There is one NameServer for each Domain; the hierarchy of NameServers follows the hierarchy of Domains. The primary purpose for NameServers is to allow applications to obtain proxies to objects across protection boundaries. Applications are allowed to look up objects in their NameServer which returns pointers to those objects; the proxy system then automatically allocates proxies for those objects so the applications can invoke methods on the objects.

An application may bind() objects only into its own NameServer, not a NameServer of any of its parent Domains. On the other hand, if a name is not found in a NameServer, the parent NameServer is consulted and so on until the kernel NameServer is reached. Bind() creates a hash table for mapping strings to objects when the first object is bound into the NameServer, so lookup() does not need to do a hash table lookup if there are no objects installed (which will be the case if there are any intermediate rings with nothing loaded into them).

7.3.4.12 ObjectProxyManager

A ObjectProxyManager instance as defined in Kernel/ObjectProxyManager.cc is responsible for allocating and freeing ObjectProxies for one Domain. It allocates the ObjectProxies out of the read-only MemoryObject for the Domain. It maintains a hash table to keep track of objects that are already proxied and returns old ObjectProxies rather than allocating new ones if possible.

The key used for the hash lookup is made up of three parts. All three parts must match before a proxy can be re-used.

1. The address of the proxied object.
2. The address of the Domain object for the Domain the proxied object is in. This is needed because of ring partitioning; there can be more than one Domain sharing the same address space so the virtual address of the proxied object is not necessarily unique.
3. The address of the proxy table for the proxied object. It is possible to have the same object treated as having more than one type: its own type and any of its parent types. If

an application gets a proxy to an object as one of the object's parent types and later gets another proxy as the object's real type, the proxy for the parent type can not be reused because then methods available only in the subclass would not be accessible[2].

The `ObjectProxyManager` limits the number of proxies that can be created for a particular `Domain` to prevent a runaway application from creating so many objects that too much memory is consumed. The number of proxies per domain are set in the MAXPROXIESPERDOMAIN constant in Memory/Domain.cc where the **`ObjectProxyManagers`** are created.

7.3.5 Initialization of Applications

The first ProcessorContext::restore() (in Includes/ProcessorContext.h) on a new **`Application-Process`** calls <Processor>ApplicationContext::restoreFromInitialContext() in ProcessorDependent/<Processor>/<Processor>ContextSwitching.s That assembler function calls the C++ assist <Processor>ApplicationContext::initialRestoreAssist() which is defined in ProcessorDependent/-<Processor>/<Processor>Context.cc. That assist function calls Domain::defaultProxy() in Memory/Domain.cc to get the default **`ObjectProxy`** to pass to the application process. In this way every application process is started with a proxy for TheSystemInterface, the instantiation of the SystemInterface class in Kernel/SystemInterface.cc.

RestoreFromInitialContext() next jumps to the beginning of the application in user mode. The initial code there is in Libraries/SystemInterface/<Processor>/crt0.s which takes the initial proxy and passes it to _Start() in Libraries/SystemInterface/Portable/_Start.cc. The first proxy call that _Start() makes is to SystemInterface::initialCall() which returns a pointer to the _ConstructProxy() function in the kernel (see section 7.3.2 for more information). _Start() passes in to initialCall() a pointer to _defaultReturn() which is defined in crt0.s. This is used as the default return address for subsequent processes that are started up in the same `Domain`. Only the first process in a application goes through _Start(); other processes start at any specified function, and if that function returns then _defaultReturn() kills the process.

Next, _Start() calls _main() in Libraries/SystemInterface/Portable/_main.cc to initialize the static constructors. This is done after initialCall() to allow the static constructors to construct

[2]Early on in the development this was more important than now because I have since changed most proxy allocations to ask the object what its real type is. The only case where this is still not true is in allocation of a proxy for an inner ring to an object in a further-out ring.

proxies. For example, if a proxied object is declared at global scope, the static constructor for that object will call the proxy constructor.

After that, version checking is done by calling SystemInterface::versions(). This is covered in section 7.9.

Next, _Start() obtains standard proxies StandardNameServer, StandardOutput, and StandardInput. The StandardNameServer is found by looking up the current process with the thisProcess() function from Includes/Libraries/SystemInterface/ThisProcess.h (which simply calls SystemInterface::getThisProcess()) and asking the proxy for the current process for the name server. StandardOutput and StandardInput are then looked up in the StandardNameServer. _Start() next takes the StandardOutput and makes it locally buffered by creating a `BufferedOutputStream` for it. BufferedOutputStream is defined in Libraries/SystemInterface/Portable/BufferedOutputStream.cc and is the only application-space subclass of ProxiableObject that is automatically included with every application.

Lastly, _Start() initializes the classes, proxy tables, and constructors. The first-class Class usage in applications is covered section 7.8. Initialization of proxy tables and constructors is done just like in the kernel because applications also support incoming proxy calls; proxy table initialization is covered in section 7.3.4.6 and the constructor initialization is covered in section 7.3.2.

7.4 Location of Ref Pointers

When a structure is returned from a method, different compilers return the value in different ways. ProxyCallAssist() needs to be aware of the differences because some methods return `Refs` which are four-byte structures.

These are the possibilities:

Value returned in registers - Some compilers are smart enough to recognize that the structure returned is only four bytes long and can optimize the code to return the value just like non-structure return values. The g++ 80386 compiler does this but only when simple structures are returned, not when a class instance like a `'Ref'` is returned.

Pointer passed in through a register - The g++ NS32332 compiler does this. Space for the `Ref` is allocated by the caller and a pointer to it is passed in through a register. ProxyCallAssist() does not need to do anything special for this case.

Pointer passed in as second parameter - The cfront C++ translator does this. Space for the `Ref` is allocated by the caller and a pointer to it is passed in as the second parameter, after the this pointer. A simple "#ifdef CFRONT" is employed in ProxyCallAssist() to skip over this extra parameter and copy it to the stack of the proxied method if the method returns a `Ref`.

Pointer passed in as first parameter - The g++ 80386 compiler does this. Space for the `Ref` is allocated by the caller and a pointer to it is as passed as the first parameter, before the this pointer. This causes problems for ProxyCallAssist() because it cannot know whether the method returns a `Ref` until after it has looked at the this pointer. The code to handle this case is enabled by the REFBEFORETHIS define. Basically, it tries to validate the first parameter as the this pointer and if it is not a valid pointer to an `ObjectProxy`, it notes that fact in the skippedRef variable and goes on under the assumption that the second parameter is the this pointer instead.

7.5 Reference Counting

All objects that are proxied must be a subclass of ProxiableObject. The most important feature of the ProxiableObject class is that it provides reference counting. A significant portion of the proxy system is dedicated to properly handling reference counting to make it transparent across protection boundaries.

7.5.1 Catching Reference Counting Methods

There are four methods relating to reference counting: reference(), unreference(), noRemainingReferences(), and the destructor. These four methods are handled specially by ProxyCallAssist(). They are detected by looking up their method numbers in the ObjectProxyCatchTable as defined in Includes/ObjectProxy.h. That table lists ObjectProxy (non-virtual!) methods used to process these methods on the `ObjectProxy` itself rather than on the proxied object: catchRef-

erence(), catchUnreference(), catchNoRemainingReferences, and catchDestructor(), respectively. These methods are defined in Kernel/ObjectProxyKernel.cc. This is what these methods do:

reference() - Instead of incrementing the reference count on the proxied object, increments the reference count on the `ObjectProxy`. (Also, the 'notReferenced' flag is cleared; see section 7.7).

unreference() - Instead of decrementing the reference count on the proxied object, decrements the reference count on the `ObjectProxy`. If the reference count on the `ObjectProxy` becomes zero, calls the catchNoRemainingReferences() method of ObjectProxy.

noRemainingReferences() - Instead of calling the destructor of the proxied object, calls the catchDestructor() method of ObjectProxy.

the destructor - Instead of calling the destructor on the proxied object, frees the `ObjectProxy` and calls unreference() on the proxied object. If the reference count on the `ObjectProxy` is not zero, prints a message and kills the current process because there was an error in the reference counting.

Since the proxied object can be outside of the kernel, the unreference() method cannot be called directly; a transfer to the ring of the proxied object must take place first. This is implemented through use of the return value of the "catch" methods: if the return value is zero, all work is assumed done. If the return value is non-zero, ProxyCallAssist() transforms the call into a normal proxied call on the unreference() method on the proxied object and proceeds.

There are two major reasons for going through this trouble:

1. To protect the inner rings from being affected by user programs. For example, if a user program called unreference() more times that it was supposed to, it could have caused the deletion of a kernel object that was still being used by other parts of the kernel.

2. To manage the `ObjectProxies` themselves. Catching these methods allows `ObjectProxies` to get freed when they are no longer needed so they can be re-used.

7.5.2 Reference Count Adjustments

Each `ObjectProxy` references its proxied object once. When an `ObjectProxy` is deallocated, its proxied object is unreferenced once. Complications arise when a proxy call returns a `Ref`; these complications may require some adjustments to the reference counts. These adjustments are primarily implemented in ObjectProxy::setProxiedObject() which is in Kernel/-ObjectProxyKernel.cc and is called whenever a proxy is allocated. Other minor adjustments are covered in sections 7.6.3 and 7.7.

First a brief background on reference counting and `Stars` and `Refs` when proxies are not in the picture: when `ProxiableObjects` are created, their reference counts begin at zero. When any method or function returns an asterisk ('*') pointer to a `ProxiableObject` (including a constructor), the reference count on the object is normally not changed. If the invoker of the method wants to refer to the returned object, it assigns the returned value to a `Star` pointer rather than an asterisk; that assignment has a side effect of calling reference() on the object that is assigned to the `Star`. As an optimization, if the method expects that its return value will always be assigned to a `Star`, the method may return a `Ref`. The semantics there is that the `Ref` pre-references the object and when the `Ref` is assigned to a `Star`, no more referencing occurs. Assigning a `Ref` to another `Ref` also has no effect on the reference count. When a `Star` or `Ref` is destructed, it automatically unreferences the object it refers to. For more information on `Stars` and `Refs` see [MCK91].

Now assume that a proxy call is being made to a method that returns an asterisk pointer to a subclass of ProxiableObject or a `Ref` (normal methods may also return `Stars` but it is discouraged because it is inefficient, and it is not supported on proxy calls). In either case a new `ObjectProxy` may be allocated or an old one may be reused because a proxy already existed for the returned object the hash table lookup matched.

returns asterisk -

new proxy - The reference count on the proxied object is incremented by one but the reference count on the `ObjectProxy` stays at zero. The application should assign the returned value to a `Star`, which will make another proxy call to reference() to reference the proxy. If the application does not reference the proxy, the proxy will

remain around until the application's **Domain** is deleted, which will deallocate all the application's proxies.

reused proxy - No reference count adjustments are needed. The proxy already references the real object. If the application assigns the returned value to a **Star**, the reference count will be incremented on the proxy.

returns Ref -

new proxy - The reference count on the proxied object has already been incremented by the **Ref**. The proxy in effect takes over ownership of that reference. The reference count on the proxy is initialized to one because the application will assume that the object it is getting is pre-referenced.

reused proxy - In this case the reference count has been erroneously incremented on the proxied object because the proxy is only supposed to reference the object once. Although inefficient, for correctness the reference count has to be decremented on the proxied object and the reference count on the proxy has to be incremented.

There is a problem in the Stable.9.23.1991 regarding adjustments to reference counts if the proxied object is outside of the kernel. In that version the kernel uses its own reference counting code to adjust reference counts on objects that are outside of the kernel.

The problem is that the definition of the REFERENCE_COUNT class is different in and out of the kernel. In the kernel, there is a machine-dependent definition that disables interrupts. The user mode code cannot disable interrupts so it uses a user-level LOCK. Thus, if the kernel uses its own REFERENCE_COUNT code to increment or decrement the reference count on the user-level object, it will not be obeying the user LOCK protocol and may violate mutual exclusion (or worse, it may get hung up because it is treating data differently in the object).

These are possible solutions I have thought of:

1. Have the kernel make an outcall to the object to change the reference count. This would be slow, and a proxy for the kernel out to that object may not already exist so a temporary proxy would need to be allocated. Also, since calls to reference() and unreference() are normally caught and processed on the proxy itself instead of on the real object, two new methods would have to be added that are somehow only callable by the kernel.

2. Use the user-level `REFERENCE_COUNT` code for kernel reference counts as well as for user-level reference counts. The user-level `REFERENCE_COUNT` code is much slower than the kernel implementation because it cannot disable interrupts; this will slow down the kernel considerably. Also, it may mean that a kernel-mode process and a user-mode process could compete for a lock for the same data (the reference count), but I do not think that will cause a problem.

3. Copy the user-level `REFERENCE_COUNT` code (including the user-level `LOCK`) to the kernel under a different name, and use that directly in the kernel when manipulating user-level reference counts. This is probably the best compromise.

7.6 Allocation, Stripping, and Validation of Proxies

The proxy trap handler automatically allocates and strips `ObjectProxies` where necessary. The primary rule that the handler follows is that proxies are always used to cross rings and they are never used within the same ring. The assumption is that pointers to `ProxiableObjects` that are passed as parameters or return values will later be used to invoke methods on those objects. In addition, some validation must be done on all pointers passed from less-trusted rings to more-trusted rings to enforce protection.

There are two major functions implementing this policy: ProxyInCallParamAssist() and ProxyOutCallParamAssist(). Both functions are used for pointer parameters and pointer return values; non-pointers are simply passed through untouched. ProxyInCallAssist() is used for parameters on incalls and return values on outcalls, and likewise ProxyOutCallParamAssist() is used for parameters on outcalls and return values on incalls; it turns out that the concerns for these pairs of cases are practically identical. In addition, there are two support functions StripProxy() and AllocateProxy() that are used by both of the major functions. All of these functions are in Kernel/ObjectProxyKernel.cc.

7.6.1 ProxyInCallParamAssist()

These are the steps done by ProxyInCallParamAssist():

1. If the pointer is a type that is not a subclass of ProxiableObject, the pointer is range-checked to ensure it points to something within the range of addresses that are accessible

by the caller. The assumption here is that the elevated-privilege callee will read or write at that address and it ought to be an address that the caller can read and write. For example, if the called method could read from a file into a buffer or write into a file from a buffer, the caller could easily read or write any value into any memory location. This check is done by looking up the ring that the address belongs to in Kernel::lookupRing() and ensuring that the ring is greater than or equal to the ring of the caller.

2. Otherwise if the pointer is of a type that is a subclass of ProxiableObject and the pointer points to a valid `ObjectProxy` for the calling `Domain`, the proxy is stripped off using StripProxy() (described in section 7.6.3). But first, if the `Domain` of the proxied object is more privileged than that of the caller, the type of the proxied object is verified to ensure that it is a type or subtype of the expected type[3] (an exception is if the proxied object is in a third `Domain` and the expected class is marked **proxiable**; for reasons see the next step). The reason for this type validation is that if the type of the object were incorrect, methods called on that object would not be the intended methods called. For example, a call to method number nine on an object that is expected to be type A would call a totally different method number nine on the object which is of type B; among other things, the parameters will be incorrect which can cause corruption of data. Since the object is coming from a less-trusted ring, compiler checks can not be trusted.

3. Otherwise if the pointer points to an address in the caller's ring and the expected subclass of ProxiableObject is one that may be subclassed outside of the callee's ring (that is, a class marked with the **proxiable** keyword), then allocate a new proxy to that object using AllocateProxy() (see section 7.6.4). There is no need to validate the type of the object, because if the type is incorrect then outcalls to methods on that object will only corrupt the less-trusted caller's space.

7.6.2 ProxyOutCallParamAssist()

These are the steps done by ProxyOutCallParamAssist():

[3]The code in Stable.9.23.1991 incorrectly compares the `Domain` of the proxied object to that of the callee rather than the caller.

1. If the pointer is a type that is not a subclass of ProxiableObject, no checking is done. There is no security violation if the more-trusted ring passes a pointer into its own space to the less-trusted ring; the less-trusted ring will simply not be able to access the memory that the pointer refers to.

2. Otherwise a distinction is made between calls from the kernel and other calls for efficiency reasons: calls from the kernel, the most common case because returns from incalls also use ProxyOutCallParamAssist(), can be handled more efficiently than other kinds of calls. Since the object or proxy in this case is coming from the kernel, a call to isProxy() can be trusted. This is a special method that returns zero from every subclass of Object except for `ObjectProxies` which return one. If the object is not a proxy, a proxy is allocated for it with AllocateProxy(). If the object is a proxy, an outcall is made to the classOf() method on the proxy to determine what the real type of the proxied object is, and StripProxy() is called to strip off the proxy and allocate a new one if needed.

   Asking the proxied object for its real type is necessary because the kernel did not do that when the object was passed in to the kernel in the first place; the expected class, which may be a superclass of the class of the real object, was used instead. For example, if ring 1 binds an object of its own into the **`NameServer`** in the kernel, the kernel's proxy is of type ProxiableObject. When ring 2 later comes along and looks up the object in the `NameServer`, the ring 1 object needs to be asked for its real class so ring 2 may make calls to all methods on that object, not just the ProxiableObject methods.

   Fine points: 1) If the proxied domain is the same as the domain being called, StripProxy() will not allocate a new proxy so it is not necessary to find the real class of the object. StripProxy() does this in Stable.9.23.1991 but it will be fixed in a later release. 2) If the kernel looks up the above example object in the **`NameServer`**, it will not get a proxy of the true type because the bind expected only a **`ProxiableObject`**. It would be nice, and it would make the implementation cleaner, if the true type of the object could be looked up when the proxy was first allocated instead of here. Unfortunately, it opens up security holes to ask the less-trusted object what its class is when allocating proxies to it from a more trusted level because the less-trusted object could lie. For example, an application object could say that its class is one that it had control of, a class that has a fake proxy

table that indicates that a proxy should be allocated for a parameter of a method when it should be an integer. Then, when the kernel makes an outcall to that method passing an integer, the application could get a proxy allocated there. 3) That security hole was not fully closed in Stable.9.23.1991. For example, an outcall from ring 0 to ring 1 that passes a proxy to an object from ring 2 will ask the object in ring 2 what its class is instead of just using the class that the ring 1 method call expects. This will also be fixed in a later release.

3. Otherwise if the outcall is not from the kernel, ObjectProxyManager::isValidProxy() is called to see if the parameter going out is a valid proxy for the calling `Domain`. If it is, the proxied object is asked what its class is as is done if the outcall is from the kernel. The difference is that the kernel does not have a proxy to that object. There is a per-process proxy out from the kernel that is allocated the first time that Process::temporaryProxy() is called and never deallocated until the process dies. This proxy is used here so there is a minimal amount of overhead. The object that the proxy points to is set by directly calling ObjectProxy::setProxiedObject() instead of going through the `ObjectProxyManager`, and reference counting is avoided to save time and complication.

   Once the class of the proxied object is obtained, StripProxy() is called to strip the proxy off and allocate a new one if necessary.

4. Otherwise if the parameter going out is not a valid proxy and the pointer points to an object in the calling `Domain`'s ring, the temporary proxy is used to ask the object what its class is and AllocateProxy() is called to allocate a proxy going out. Unlike in ProxyInCallParamAssist(), the class of the object going out does not need to be marked with the **proxiable** keyword because the caller here is more trusted than the callee.

7.6.3 StripProxy()

The job of StripProxy is to strip a proxy off and allocate a new one if necessary. If the called `Domain` is the same as the `Domain` of the object that the proxy is for, the proxy need only be stripped off; if the called `Domain` is different, StripProxy() also calls AllocateProxy() to allocate a new one.

In addition, if StripProxy() is being called for the return of a method that returns a `Ref`, the caller would have erroneously incremented the reference count on the proxy. To counteract that, StripProxy() decrements the reference count on the original proxy and increments the reference count on the new proxy or on the real object.

If the proxy that is being stripped is one that is being returned by a method and the proxy has never been referenced, StripProxy() deallocates the proxy. See section 7.7 for more details.

7.6.4 AllocateProxy()

The reason that AllocateProxy() exists is to keep track of proxies that are allocated for incoming parameters so that they may be freed on return from the proxy call. AllocateProxy() allocates a proxy by simply calling ObjectProxyManager::allocate(), but after that it does bookkeeping to keep track of the allocated proxies. For more information on freeing proxies that do not get referenced, see section 7.7.

7.7 Freeing Not-Referenced Proxies

As I have shown, proxies are sometimes automatically allocated for parameters to a proxy call. The normal semantics without proxies involved is to leave the reference counts alone on an object passed to a method unless the method wants to retain a pointer to the object; usually an object is just used temporarily so the net effect is no change to reference counts. When a proxy is allocated to pass to such a method, the proxy must to be deallocated when the method returns from the proxy call because the proxy will most likely not be used again.

Every time a proxy is allocated by the proxy trap handler, AllocateProxy() calls ObjectProxy::setNotReferenced() which is in Includes/ObjectProxy.h. That method sets a bit in the `ObjectProxy` called notReferenced if the reference count of the allocated proxy is zero. "Not referenced" is not to be confused with "unreferenced"; "not referenced" means that the proxy has never had reference() called on it and "unreferenced" means that it has had unreference() called on it. If the reference count was zero then setNotReferenced() returns true, and then if the proxy is being allocated for a parameter going in to a method call (as opposed to a return value), a pointer to that proxy is saved in the returnFlags.

When the proxy call returns, ProxyReturnAssist() goes through the proxies that were saved in the returnFlags and calls ObjectProxy::freeIfNotReferenced() in Kernel/ObjectProxyKernel.cc for each one. If the notReferenced bit in the proxy is still turned on, then freeIfNotReferenced() deallocates the proxy. It also decrements the reference count on the proxied object because the proxies reference their proxied object. It deliberately avoids calling unreference() on the proxied object because the object should not be deleted if the reference count becomes zero; if the proxy had not been in the way of the call then the object would not have been deleted even if the reference count had started out at zero so the same thing should hold if the proxy is in the way.

There is one other place where a proxy is freed if it is not referenced. If a proxy is being stripped off for a return value from a method call, StripProxy() calls freeIfNotReferenced(). An example of this is if a proxied method in ring 1 uses another proxy call to the kernel to get a proxy to an object in the kernel, and then immediately returns that value[4]. In that case the proxy was only used temporarily and need not remain around.

7.8 First-class Classes

The hierarchy of Classes in *Choices* is mirrored by a hierarchy of objects known as first-class classes. Every subclass of Object is represented by an instance of the class called `Class`. This most important use of this hierarchy is run-time type checking.

7.8.1 Hierarchy of Class Hierarchies

The hierarchy of classes in the kernel is a simple tree rooted in class Object; only single inheritance is supported. Since the class hierarchy can now be extended outside of the kernel, the first-class class hierarchy must also be extended. The extension can be done independently in each `Domain`, so there is a hierarchy of `Class` hierarchies that follows the `Domain` hierarchy.

All `Class` objects are located in the kernel but they keep track of the ring they belong to, and their names are installed in the **`NameServer`** for the `Domain` that they belong to. The implementation requires an application to copy `Class` objects that represent classes in a further-

[4] I am not certain that this call to freeIfNotReferenced() is always the right thing to do, but I do not think it will cause any problems, and it solved a problem like the example.

in ring in order to subclass a class from the further-in ring. Copying of `Class` objects is handled automatically and discussed further in section 7.8.2.

When a Class is first bound into the `NameServer` through the Class::bind() method in Kernel/Class.cc, it first looks in the `NameServer` to see if there is already a `Class` by that name. Most likely the `Class` will be found in the next inner ring. If the `Class` is found, Class::bind() keeps a pointer to it in the _real pointer[5].

The Class::isClass() and Class::isASubClassOf() methods follow the _real pointers until the rings of the compared `Classes` are identical. In this way type checking can compare classes represented in different rings.

The Class::sibling() method also follows the _real for use in printing the entire hierarchy of classes. If there are no more siblings for a `Class` in a particular ring, Class::sibling returns the first child of its parent's _real `Class`. For example, if an application defined a MySemaphore class as a child of Semaphore, MySemaphoreClass->sibling() would return a child class of Semaphore in the kernel such as BinarySemaphore[6].

7.8.2 Initializing Classes

`Class` objects are initialized through use of `ClassInstaller` which is defined in Common/-ClassInstaller.cc. `ClassInstallers` are statically constructed objects that contain enough information to later create a `Class` and which install themselves in a list. ClassInstaller::install() goes through the list and creates the `Classes`.

An implementer of a new class Foo must manually add a DEFINECLASS() (defined in Includes/Common/ClassInstaller.h) macro to his source file. This macro creates a `Class-Installer` for FooClass. It also defines a Foo::classOf() method and the global variable FooClass.

The entire source code for some classes are copied into applications, including the `DEFINE-CLASS()` that defines their `Classes`. In this way every application is initialized with a `Class` at least for Object, ProxiableObject, InputStream, OutputStream, and BufferedOutputStream.

In addition, Proxify++ generates an invocation of the COPYCLASS() macro (also defined in ClassInstaller.h) for every proxiable class in $(MODULENAME)STUBS.cc. COPYCLASS()

[5]There is no particular reason it has to use the _real pointer, another variable could have been added to Class instead.

[6]Stable.9.23.1991 incorrectly returned the child of the parent of its own _real `Class`.

is exactly like DEFINECLASS() except that it does not define the classOf() method. These COPYCLASS() invocations automatically make it possible to subclass proxiable classes in applications.

7.8.3 ClassConstructor

There is a method called Class::constructor() that can instantiate an object of the class that the `Class` represents. This does not work for all classes, only those that have provided a constructor; this is used primarily in the filesystem.

Because the object to be constructed might not necessarily be in the kernel, and because all `Class` objects are in the kernel, Class::constructor() cannot always call the constructor function directly. Instead, the kernel needs to make an outcall through a proxy to call the constructor. ClassConstructor in Common/ClassConstructor.cc is the class that is proxied to do the job. ClassConstructor is copied into every application. ClassInstaller::install() creates a `ClassConstructor()` for every class that provides a constructor function as the fourth parameter of DEFINECLASS().

7.9 Version Checking

The version checking subsystem keeps track of a version string for every proxiable class. During initialization of an application (covered in section 7.3.5), each proxiable class that the application uses is checked to ensure that the interface the application expects matches the interface that the kernel (or other proxied load module) provides. This catches many common development errors.

To prepare for that, Kernel::main() finds the list of provided versions by calling ProvidedVersionInstaller::list() in Common/VersionInstaller.cc. That list was created using static constructors in Configure/System/<Machine>/PROVIDEDVERSIONS.cc which is automatically generated by the makefiles. That file includes a call to the macro PROVIDEDVERSION() which is defined in Includes/Common/VersionInstaller.h for each proxiable class. Each time an alternate header file changes, the makefiles generate a new version string for that class using the output of the 'date' command. Header files that do not change use the old version strings. The

static constructors for ProvidedVersionInstaller save the date string and install themselves in a list.

Kernel::main() passes that list to VersionObject::install() in Kernel/VersionObject.cc which creates a `VersionObject` for each element and installs each `VersionObject` in the `NameServer`.

The versions on the application side are similar but there is an extra level of indirection. Proxify++ generates a call to the REQUIREDVERSION() macro inside every generated header file. That macro is defined in VersionInstaller.h. Wherever the generated header files are included, REQUIREDVERSION() creates a statically constructed `RequiredVersionInstaller` object which installs itself in the RequiredVersion list so that only classes that are actually used by the application have their versions checked. Because the header file may be included from many different source files in the application, to save space the version string is not included in the `RequiredVersionInstaller` but instead there is a pointer to a `VersionInstaller` object for that class. One `VersionInstaller` object for every available class is created through the the VERSION() macro in Configure/System<Machine>/SystemVERSIONS.cc. That file is compiled into the library that is linked with applications and is exactly the same as the PROVIDEDVERSIONS.cc that is linked with the kernel except for the macro names[7].

In addition, a VERSION() is generated for the files that are compiled with both applications and the kernel in Configure/System/<Machine>/TableOnlyProxyVersion.cc. This file is updated whenever any of the files listed in the $(TABLEONLYPROXYINCLUDES) make variable change (this list includes files that are copied into applications). This contents of this file is copied into SystemVERSIONS.cc and from there into PROVIDEDVERSIONS.cc.

During the initialization of applications, the `RequiredVersionInstaller` list is given to VersionObject::check() which looks up each version in the list from the `NameServer` and warns if the version is not found or if the versions do not match.

[7]The name "System", the $(MODULENAME) make variable, is included in the filename to distinguish it from other VERSIONS files of other proxiable modules that an application uses. The g++ compiler could not link in two files with the same name because it generates a symbol in each file including the filename that becomes multiply defined at link time.

7.10 Running Applications in Any Ring

I have provided a user interface to create new **Domains** in any ring outside of the kernel and to run applications in any **Domain** outside of the kernel. The following commands can be used at the "Enter application" prompt, as implemented in Kernel::getMemoryObjectOfInitialApplication:

newDomain - create a new **Domain**. 'newDomain' takes two parameters: the name or number of the parent of this **Domain** in the **Domain** hierarchy and a name for the new **Domain**. The newly created **Domain** will be in the next ring further out from the parent. If just a number is used for the parent **Domain**, where the number is < APPLICATIONRING, then the default **Domain** for that ring is used. Default **Domains** are created for the kernel and each intermediate ring at boot time. If a number is not used, then the name is looked up in the **NameServer**. The newly created **Domain** is put into the **NameServer** under the given name. Examples: "newDomain 0 dom1" creates a new **Domain** in ring 1 called 'dom1' and "newDomain dom1 dom2" creates a new **Domain** on top of that in ring 2 called 'dom2'.

load - load a new application into an existing **Domain** and start a process in it but do not wait for the process to complete. The first parameter must be either a default **Domain** number to load into or a name of a **Domain** that was created with 'newDomain'. The second parameter is the complete path name of the executable. Any more parameters will get passed as argc/argv to the program. Another "Enter application" prompt will come back immediately because 'load' does not wait for the process to finish. Example: "load 1 load1".

run - run a new application in a new **Domain**, start a process in it, and wait for the process to complete. This is much like just entering the name of the application except it allows you to specify a different domain that you want the application to run above. It takes a first parameter of the **Domain** that will be the parent of the new **Domain** created for the application; the parameter can be a number or a name. The second parameter is the complete pathname for the executable. Any more parameters will be passed as argc/argv to the application. Example: "run 2 /Bin/timings fast".

7.11 Summary

This chapter has given implementation details on the proxy subsystem of *Choices*. The chapter has provided information that is useful to those who desire to thoroughly understand how the subsystem works and to those who desire to modify it. The implementation supports the strict enforcement of protection, provides language transparency, performs adequately, and provides for more than two protection levels.

Chapter 8

Performance

In this chapter I discuss the performance of the proxy subsystem. All of the timings were taken by executing calls at least 10,000 times and dividing the total time taken by the number of times the calls were made. Thus the times are average elapsed time, including any repetitive system overhead such as clock interrupts.

The timings on *Choices* were taken with the 'timings' application program which tests and times all combinations of calls between three rings. The timings on Unix[1] [Bou83] were taken with a similar Unix program. Not all of the results are included here; I have summarized the interesting ones. *Choices* timings were not taken on the Stable.9.23.1991 version, they were taken on a version that had a few problems mentioned in chapter 7 fixed.

Timings taken by this scheme are not precisely consistent and often vary by 5%. The measurements given here are the average of two runs.

8.1 Comparison to Unix System Calls

Table 8.1 shows a comparison between simple Unix system calls and proxy calls. The AT&T6386 PC is based on a 20MHZ Intel 80386 processor and the Multimax is based on 10MHZ National Semiconductor 32332 processors. The Multimax is a shared memory multiprocessor, but only one processor is used for these timings.

[1] Unix is a trademark of AT&T.

	AT&T6386	Multimax
Unix getpid()	118μs	97μs
Unix time(0)	118μs	171μs
Null proxy call	84μs	136μs

Table 8.1: Simple Unix system calls versus *Choices* null proxy calls.

The null proxy call is a call to TheSystemInterface->null() in the kernel which takes no parameters and immediately returns. There is no exact equivalent in Unix, but getpid() and time() are simple Unix calls that should need to do little extra work. As the table shows, proxy calls and Unix system calls take roughly the same amount of time.[2]

By contrast, local C++ virtual function calls on both machines take 6μs. Crossing the protection boundary into the kernel takes more than an order of magnitude longer on both operating systems. That cost must be weighed against the advantages of having protection. If an application requires high performance, it may have to be designed to minimize the number of times that protection boundaries are crossed.

8.2 Calls Between Rings

Table 8.2 shows the times for null proxy calls between different rings.

	AT&T6386	Multimax
Incall to kernel	84μs	136μs
Outcall from kernel	185μs	256μs
Incall to non-kernel	192μs	339μs
Outcall from non-kernel	195μs	341μs

Table 8.2: Null proxy calls between rings.

Calls to rings other than the kernel take approximately twice as long as calls to the kernel. Since calls into the kernel are the most common case, they have been optimized the most. Calls

[2] I have found that the Multimax tends to be about 30% slower than the AT&T6386 so the Multimax getpid() timing is a bit surprising. I included time() to show that perhaps the Multimax version of Unix is optimised for getpid().

to non-kernel rings first trap to the kernel and perform the normal operations of kernel calls as well as the following additional operations:

1. Different user mode stacks are needed for each ring outside of the kernel, so the correct one is located and set up.

2. C++ compilers assume that some processor registers are not modified by function invocations, but a call to user mode cannot be trusted to leave those registers unmodified. In order to prevent detrimental effects on the kernel caused by the registers being modified, the registers are saved before switching to user mode and restored upon return from user mode.

3. Memory to be protected from the called ring is made inaccessible and memory to be available from the called ring is made accessible. This is done differently on the AT&T6386 and Multimax, as explained in section 4.2. On the AT&T6386, this is accomplished by changing data segments which is a very fast operation. On the Multimax, it is accomplished by loading a different page table. This is only necessary when switching between two different non-kernel rings; page tables can contain different protection for kernel and user modes, and user mode is used for all non-kernel rings. The extra 80μs for calls between two non-kernel rings on the Multimax is due to changing to a new page table before the call and restoring the old one afterward. This increase amounts to only 30% above outcalls from the kernel.

4. The method is executed in user mode, and when the method returns it traps back to the kernel.

An alternative to using intermediate rings would be to run server code in an entirely separate Domain and send messages to the server process, similar to remote procedure calls. This alternative inherently has extra overhead of passing messages through either copying the data or some other way of sharing. That overhead is not present with intermediate rings because data in the outer rings is directly accessible to called methods on incalls. In addition, sharing of address space makes rings easier to incorporate into a language such as C++ which assumes one large address space.

For the rest of the timings in this chapter there is no fundamental difference between the Multimax and AT&T6386 so I will concentrate on the AT&T6386 only.

8.3 Parameters

The proxy trap handler processes every parameter to a proxy call and thus takes extra overhead for every parameter. Table 8.3 shows the extra time taken for different kinds of parameters.

	above	Incall kernel	Outcall kernel	Incall non-krnl	Outcall non-krnl
0. No parameters	-	84μs	185μs	192μs	195μs
1. One integer	0	25μs	14μs	14μs	14μs
2. Two integers	1	6μs	6μs	6μs	6μs
3. Non-ProxiableObject pointer	1	21μs	11μs	23μs	11μs
4. Called-ring ProxiableObject pointer	3	45μs	31μs	45μs	33μs
5. Caller-ring ProxiableObject pointer	3	221μs	190μs	220μs	560μs
6. - additional for new proxy	5	296μs	290μs	289μs	285μs

Table 8.3: Times for proxy call parameters on AT&T6386.

The "above" column indicates the line number that this line should be added to for the total time. Apply the addition recursively to find the total time for a call. For example, to find the total time for a call with a single called-ring ProxiableObject pointer parameter, add a column from line 4 to the same column from lines 3, 1, and 0; for an incall to the kernel that is 175μs. To find the time for subsequent parameters use line 2 instead of line 1; for example a second ProxiableObject parameter on an incall to the kernel adds 72μs for a total of 247μs.

Here is a discussion of each of the lines in the table:

0. Copied from table 8.2.

1. The existence of any parameters requires some set-up time. A call to the kernel with no parameters is specially optimized, so the first parameter for calls to the kernel adds more than the first parameter for calls to other rings.

2. Each additional non-pointer parameter adds an insignificant amount of time.

3. Incall pointer parameters that do not point to subclasses of ProxiableObject are range-checked to ensure that the pointer is an address that the caller has access to. This checking does not occur on outcalls.

4. A caller passing a parameter that points to an instance of a subclass of ProxiableObject in the called ring implies that a proxy is being passed. Extra time is taken to verify that the proxy is valid for the caller. In addition, incalls verify that the type of the proxied object is an expected type.

5. A caller passing a parameter that points to an instance of a subclass of ProxiableObject in its own ring implies that a proxy to the object must be allocated for the called ring. Proxy allocation first looks up the proxied object in a hash table and then re-uses a proxy if one already exists. The times in line 5 of the table are for re-use of a proxy. The extra time for an outcall from a non-kernel ring is caused by the kernel trap handler making an extra outcall to the calling ring to find out the actual type of the object being passed. This is necessary because the type may be a subtype of the expected type, and a later incall using that proxy will need to know the exact type. This is discussed in more detail in section 7.6.2.

6. If a proxy did not already exist, a new one is allocated on the way in to the proxy call and freed on the way out. This additional time is the same for all the kinds of calls and includes installing an element into the hash table and removing it again. A general purpose hash table implementation is used here which could probably be improved. The time gets even longer with this hash table implementation if the hash table slot was already in use: an additional 456μs is taken in that case to create and delete a new slot.

8.4 Returning Proxies

Returning pointers to subclasses of ProxiableObject implies returning proxies. Presumably a proxy that is returned will be used multiple times to invoke methods; thus the returning of proxies will be relatively rare and the length of time that they take might not significantly affect overall performance of an application.

	above	Incall kernel	Outcall kernel	Incall non-krnl	Outcall non-krnl
7. Returner-ring pre-existing proxy	0	233μs	250μs	578μs	251μs
8. - additional for new proxy	7	381μs	483μs	490μs	496μs

Table 8.4: Times for returning ProxiableObject pointers on AT&T6386.

Table 8.4 shows times for returning pointers to subclasses of ProxiableObject. Here is a discussion of each line:

7. The return value of an incall is very much like a parameter of an outcall, and the return value of an outcall is very much like a parameter of an incall. Thus returning a pointer to an object in the returner's ring is much like passing a pointer to an object in the caller's ring as shown in line 5 of table 8.3.

8. One difference between parameters and return values is that proxies allocated for parameters are automatically deallocated when the call returns; that is not the case for return values. Since I do timings by repeated calls and since an application is limited in the number of proxies that may be allocated for it, the times for this line include an extra call to explicitly deallocate the proxy. The extra call is reflected in additional time over the 290μs of line 6 in table 8.3. The additional 90μs for incalls to the kernel and 200μs for other calls are approximately the same as times for null calls.

Chapter 9

Conclusions

This research has developed and studied a mechanism to support object-oriented hierarchies across protection boundaries. The mechanism strictly enforces protection and yet is as transparent as possible to the programmer to allow a uniform object-oriented programming model. The mechanism has adequate performance so as to not preclude its use.

The mechanism is transparent: it automatically and transparently inserts and removes special objects called proxy objects which are used to cross a protection boundary and to invoke methods on individual objects on the other side of the boundary. Applications not only can interface with system objects as if they are local objects, they can also incrementally modify the behavior of the system objects for their own use through inheritance. The operating system can also take advantage of the inclusion polymorphism of type hierarchies to define interfaces through which it can call out to less-privileged subtypes.

The performance of crossing protection boundaries is adequate: method invocations into the kernel with no parameters takes approximately the same amount of time as a Unix system call, and the extra time taken for allocation and removal of proxy objects for parameters and return values is not detrimental because those operations only occur on a relatively small percentage of calls in a typical application. My implementation also uses intermediate rings to support structuring of the operating system into kernel and non-kernel portions; calls to the intermediate rings take more time than calls to the kernel, but they are faster than calls to separate address spaces because message passing is not required.

The study of this mechanism led to an analysis of the fundamental problems caused by splitting type hierarchies and inheritance hierarchies across protection boundaries. The analysis is independent of language, system, and protection model. The analysis revealed the precautions that must be taken to guard against protection violations in an object-oriented system with protection boundaries. The analysis also showed that in the general case an object must be able to be split across the protection boundaries, and that the portion of the object belonging to a child in an object-oriented hierarchy should delegate or preferably forward unrecognized method calls to the parent portion of the object on the other side of the boundary.

The remainder of this final chapter contains a summary of conclusions, a summary of problems caused by the implementation constraints, and suggestions for future work.

9.1 Summary of Conclusions

The following is a summary of conclusions that can be drawn from this research:

1. Extending object-oriented hierarchies beyond the kernel of an operating system is worthwhile because it provides a uniform computation model to programmers of object-oriented applications. It is also worthwhile for structuring the operating system itself into portions inside and outside of a minimal kernel, as demonstrated by the successful operation of the filesystem outside of the *Choices* kernel.

2. A ring structure with shared address spaces fits well with the introduction of protection boundaries into an object-oriented language that is based on one large shared address space. The shared memory semantics that this provides is nearly transparent to the programmer, deviating only where transparency would cause protection violations. In addition, the time that it takes to make calls to intermediate rings compares favorably to the time that it would take to call to a separate address space because of the necessity of message passing when separate address spaces are employed. Performance of calls to intermediate rings is even better when the hardware is powerful enough to render changing page tables unnecessary.

3. Using proxies to control the crossing of protection boundaries in an object-oriented system is practical and efficient. A method call into the kernel using a proxy takes approximately

the same amount of time as a system call in Unix. Automating the allocation, validation, and stripping of proxy parameters and return values makes the proxies practically transparent to the programmer and eliminates sources of errors that are caused by hand-coding those tasks. This automation can reliably assist in satisfying the following requirements which are applicable to any system that provides object-oriented hierarchies across protection boundaries:

(a) Untrusted subjects should be permitted to invoke methods only on objects to which they have obtained rights. This is satisfied by proxies only giving the capability to invoke methods on specific objects. In addition, new proxies are only allocated for objects that are returned or passed in through other proxy calls.

(b) Untrusted subjects should be permitted to invoke only selected methods on the objects to which they have access. This is satisfied in my system by marking a method with the **proxiable** keyword.

(c) Objects passed in as parameters to a method call across protection boundaries should be validated to ensure that the caller has the right to use those objects. This is satisfied in a proxy-based system by validating that parameters that refer to protected objects are proxies that the caller has the right to use. Also, stripping the proxies off so the called method does not have to deal with those proxies and/or allocating new proxies for the object parameters is required if proxies can only be used by one domain.

(d) If objects passed in as parameters to a method call across protection boundaries do not trust the caller, the type of the objects should be validated to ensure they are of a type that the called method expects. This may be automatic in a language that has run-time type checking, but it needs to be added to languages like C++ that have no run-time type checking.

(e) Untrusted subjects should be permitted to create only objects that belong to subjects that have chosen to allow it, and untrusted subjects should only be permitted to delete objects that they have created. In my system, creation is controlled by only allowing creation of proxied objects that have their constructors marked with the

proxiable keyword, and deletion is controlled by only deleting proxied objects when their reference counts become zero.

(f) The object types that can be polymorphically replaced by an object of an untrusted subtype should be restricted to only selected types. This is satisfied in my system by marking a class definition with the **proxiable** keyword.

4. The portions of an object that belong to a superclass and subclass in an implementation hierarchy where the superclass is more trusted than the subclass should be handled in one of two ways:

   (a) If a superclass does not need to cause side effects in its own protection domain, it is better to copy the superclass code into the subclass' protection domain and to place the entire object at that combined level. Unfortunately this is potentially problematic in a language like C++ because the same class description has to be used for both the copy of the superclass and for proxies to the original superclass.

   (b) If the superclass must continue to be able to cause side effects at its more powerful trust level, the portions of the object belonging to the superclass and subclass must be protected at the different trust levels. Method calls that the subclass does not redefine should be delegated or forwarded to the superclass portion of the object. Forwarding is preferred to delegation in this case because delegation implies that further method calls on the same object could call back out to the less trusted subclass portion of the object; that should only happen in a controlled manner.

5. The best design of an object-oriented hierarchy in the absence of protection boundaries may not be the best design when protection boundaries are introduced. Crossing protection boundaries is costly, and a class hierarchy may need to be redesigned to minimize the number of times that the boundaries are crossed.

9.2 Problems Caused by Implementation Constraints

The two major constraints on my implementation were the use of the C++ language and the use of traditional processor protection hardware.

These are the problems caused by the choice of C++ as an implementation language:

1. The lack of run-time type information and type checking causes the biggest problems. Type checking needs to be performed when protection boundaries are crossed because callers cannot be trusted to obey compile-time type checks. First-class classes were created to represent the type hierarchy at run-time, and Proxify++ was created to generate the type information for methods that can be called across protection boundaries.

   The lack of run-time type information also makes it impossible for the run-time system to know the format of all parameters that are passed across protection boundaries. A consequence of this is that pointers to arbitrary data should not be included in any structures passed as parameters to privileged methods because the pointers are not checked to ensure they point to memory that the caller has the right to access.

2. The lack of garbage collection in C++ causes difficulties for cleaning up protected objects; non-privileged users must not be allowed to directly delete protected objects. To get around these difficulties, reference-counting was used; this works, but garbage collection is more convenient and reliable.

3. The use of header files to describe class interfaces and implementation details (that is, member variables) in C++ causes problems. Proxify++ gets around some of the problems by generating an alternate header file with non-accessible methods and implementation details removed.

   However, there is still a problem when the same header file is needed to interface with a class that is both copied into a ring and accessed through a proxy. In that case an alternate header file cannot be used when going through a proxy so the class interface has to be restricted so that all **virtual** methods are **proxiable**, and users of the proxy have to be careful to not try to access any of the member variables which do not exist in the proxy.

4. Since C++ implements parent and child portions of objects in contiguous memory, splitting an implementation hierarchy across protection boundaries requires simulation of inheritance by forwarding. A delegation-based language such as Self would have an easier time with this. Simulating true delegation in C++ requires changing the interface to pass an additional parameter referring to the child object.

5. Since C++does not allow the separation of implementation hierarchies and type hierarchies, the generality of my research is reduced. Languages that do allow the separation may have additional problems that have not been uncovered by this research.

The use of traditional processor protection hardware for this implementation causes reduction in performance. Protection hardware that had more than two protection levels directly supported would be the fastest. The requirement of switching page tables when switching rings on processors without segmentation is particularly inefficient.

9.3 Future Work

Here are some suggestions for possible extensions to this research:

1. More services that are currently a part of the *Choices* kernel could be moved out into an intermediate ring. The networking code is an excellent candidate for the next large portion to move.

2. Cross-calls to separate address spaces could be implemented and evaluated. Many of the techniques of remote procedure calls would have to be employed. Work is already in progress to implement remote procedure calls in *Choices*, and that work could also be applied to invoke methods on objects in different address spaces on the local machine.

3. The techniques of this research could be applied to other languages and operating systems. Applying the techniques to an object-oriented language with dynamic run-time type checking and garbage collection such as CLOS would be particularly interesting so it could be compared to my system which uses substitutes for those features because they were not a part of C++.

Appendix A

Source Files

This appendix lists all *Choices* source files related to the proxy subsystem in some way. The list does not include the source files of Proxify++itself; those are discussed in section 7.1.3.

Most of the '.cc' files listed here have corresponding '.h' files under the 'Includes' directory. The '.h' files that do not have corresponding '.cc' files are included explicitly in this list.

If the file is discussed in more detail in another section, the section number is in parentheses after the filename.

Applications/Examples/pk.cc - An application to peek or poke a specified address to verify whether or not an address is protected from applications. Also tests outcalls by using Process::setFaultHandler() and creating its own FaultHandler which is called by Domain::basicRepairFault when there is a page fault.

Applications/Examples/timings.cc - An application that tests and times all combinations of incalls and outcalls using the TimingInterface in the kernel. If the intermediate Example load module (load1) is loaded into ring 1, also tests and times calls to and from that ring using the SubTimingInterface.

Applications/Examples/twoproc.cc - Example application that creates another process and a semaphore. Twoproc declares the `Semaphore` at global scope rather than creating it using **new** and in so doing tests _PostConstructProxy() and method stubs.

Applications/FiSh/Hierarchy.cc - The "hierarchy" command in FiSh prints out the ring number that `Classes` belongs to if they are not in the kernel.

Applications/LoadableExample/SubTimingInterface.cc - A subclass of TimingInterface (which is in the kernel) to test and time incalls and outcalls to and from an intermediate ring. Used by the 'timings' application.

Applications/LoadableExample/TestObject.h - Simple test object that is added to the `NameServer` for ring 1 and made available to applications.

Applications/LoadableExample/main.cc - The main part of the Example loadable module. Creates a `TestObject` and a `FileSystemInterface`. The source for the filesystem in this module is taken directly from the FileSystems directory, the same one that is used for the kernel. The process exits but the code remains loaded until the `Domain` is deleted at system shut down time.

Common/ClassConstructor.cc (7.8.3) - Ensures that constructors called via Class::constructor() are called in the correct ring of protection.

Common/ClassInstaller.cc (7.8.2) - Defines `Classes`.

Common/ConstructorInstaller.cc (7.3.2) - Defines constructors that are callable through proxies.

Common/InputStream.cc - Defines `InputStream` abstract superclass. Copied into all applications.

Common/Object.cc - Defines the base class of most objects in *Choices*. Copied into all applications. Contains default definition of the method isProxy() which returns zero. Also contains identity() method which returns the this pointer for use in case a proxy is in the way of an object[1].

Common/OutputStream.cc - Defines `OutputStream` abstract superclass. Copied into all applications so printf() and other methods are processed in user space.

Common/ProxiableObject.cc - Abstract superclass of all objects that need reference counting. All proxied objects need reference counting, so they must be instances of subclasses of this class.

[1]Now that proxies are always stripped off within a `Domain` and proxies to a particular object are reused, identity() should no longer be necessary.

Common/ProxyTable.cc (7.1.2, 7.3.4.6) - Definition of the proxy table. Also the definition of ProxyTableInstaller which is used to locate and initialize proxy tables.

Common/VersionInstaller.cc (7.9) - Defines version strings for proxied classes that are required and provided.

Configure/<Machine>Common.mk (7.2) - Defines machine-specific make variables and rules.

Configure/Applications/ApplicationsCommon.mk (7.2) - Defines make variables and rules that are useful for all applications and loadable modules. Puts different directories in the $(COMMONHVPATH), includes different libraries, and includes ProxifyCommon.mk for loadable modules; that is, if $(MODULENAME) is set.

Configure/ChoicesCommon.mk (7.2) Defines make variables and rules that are useful for all of *Choices*.

Configure/LoadableExample/ExampleCommon.mk - Example makefile for a loadable module.

Configure/System/<Machine>/LoaderDirectives (7.3.4.5) - This is the place that the KernelReadOnlyStart and KernelReadOnlyEnd variables are defined for both Att6386 and Multimax. On compilation systems that do not have COFF link editors, this mechanism will not work. Those variables could then be defined using assembler globals or something similar.

Configure/System/CommonSources.mk - Lists source files that are common to kernel and applications.

Configure/System/FileSystemsCommon.mk - Lists sources for the filesystem. These are here and not in SystemCommon.mk because the filesystem is included in both the kernel and in the Example loadable module.

Configure/System/ProxifyCommon.mk (7.2) - Makefile included by makefiles of modules that want files proxified.

Configure/System/SystemCommon.mk - Makefile that defines the portion of *Choices* kernels that are common across different machine types.

Includes/Libraries/SystemInterface/REFERENCE_COUNT.h (7.5.2) - Definition of reference counts for applications. This is portable, unlike the machine-dependent one in the kernel. However, it is slower because applications cannot disable interrupts to aid in mutual exclusion. Instead, a user-level LOCK is used.

Includes/Libraries/SystemInterface/ThisProcess.h (7.3.5) - Definition of thisProcess() for applications. Uses the `SystemInterface` to get the information from the kernel via a proxy call.

Includes/MachineDependent/<Machine>/<Machine>CPUConfiguration.h (7.3.4.1) - Place where APPLICATIONRING is defined.

Kernel/CPU.cc (7.3.4.3) - Has a method setRingRange() which is called at initialization time. May be overloaded by processor-dependent subclasses; if not, the method does nothing.

Kernel/Class.cc (7.8) - First-class classes. Supports hierarchy of class hierarchies across rings.

Kernel/ConstructorDescriptor.cc (7.3.2) - Descriptor for proxiable constructors.

Kernel/Kernel.cc (7.3.4) - Definition of the Kernel class which is primarily a catch-all class for things that would otherwise be global; there is only one instance of this class. It is intimately involved with the initialization of the proxy subsystem, and it keeps track of the address ranges of rings.

Kernel/NameServer.cc (7.3.4.11) - The name server class. The `NameServer` hierarchy follows the `Domain` hierarchy.

Kernel/ObjectProxyKernel.cc (7.3) - The corresponding header file is Includes/ObjectProxy.h. Defines most of the ObjectProxy class and the machine-independent portion of the proxy trap handler.

Kernel/ObjectProxyManager.cc (7.3.4.12) - Manages allocating and freeing proxies. For every `Domain` there is one `ObjectProxyManager`.

Kernel/ObjectProxyVtable.cc (7.3.1) - Contains the ObjectProxy constructor and isProxy() method. These are in a separate file so they can be made to be readable outside of the kernel.

Kernel/Process.cc (7.3.3, 7.6.2) - Definitions of processes. Has some support for proxies, including keeping track of the current `Domain` and maintaining a temporary proxy for each process.

Kernel/ProxyTableCopy.cc (7.3.4.6) - Definition of the kernel copy of `ProxyTable`.

Kernel/SystemInterface.cc (7.3.5) - Every application is started with a proxy to the sole instance of this class. Contains miscellaneous system calls that do not fit well in other proxiable classes.

Kernel/TimingInterface.cc - Class for testing and timing incalls into and outcalls from the kernel. Used by the 'timings' applications.

Kernel/TwoLevelPageTable.cc (7.3.3) - AddressTranslation used by both Att6386 and Multimax, although it is not portable to all architectures. Contains support for multiple ring protection when the "MULTIRINGTRANSLATION" define is turned on.

Kernel/VersionObject.cc (7.9) - Class to enable installation of version checking information into the `NameServer` hierarchy.

Libraries/GeneralPurpose/Portable/HashTable.cc - Contains general purpose hashing functions. Used both by the Proxify++ tool and by the `ObjectProxyManager`. HashTable was extended for the `ObjectProxyManager` to allow hashing on fixed-length arbitrary data and optimized to improve performance of proxy allocation.

Libraries/SystemInterface/<Processor>/Constructors.s (7.3.2) - Definition of proxy constructor stubs.

Libraries/SystemInterface/<Processor>/crt0.s (7.3.5) - Application processes begin executing at the beginning of this file.

Libraries/SystemInterface/Portable/BufferedOutputStream.cc (7.3.5) - Application-level subclass of OutputStream that buffers output to the stream until the buffer is full or a newline or a flush().

Libraries/SystemInterface/Portable/LOCK.cc - A user-level lock that is not portable but should be correct for Att6386 and Multimax. Assumes atomic increment and decrement. Avoids making a system call if there is no contention for the lock. The algorithm is not "fair"; that is, it has possibilities of starvation.

Libraries/SystemInterface/Portable/_Start.cc (7.3.5, 7.3.2) - Machine-independent portions of application initialization and other miscellaneous machine-independent functions including _PostConstructProxy() for use by all applications.

MachineDependent/<Machine>/<Machine>Kernel.cc (7.3.4.2) - Defines variables for the `Kernel` that define the start and length of rings.

Memory/AddressTranslator.cc (7.3.3) - The parent class for MMUs. The activate() and basicActivate() methods were extended to accept a parameter of the context of the process being executed. The processor-dependent basicActivate() uses that to find out the current ring number of the process being activated if the MMU implements the ring protection.

Memory/Domain.cc (7.3.4.10) - Representation of address spaces. Extended to support multiple rings. Maintains a pointer to an `ObjectProxyManager` and a pointer to the default `ObjectProxy` for the `Domain`.

Memory/DummyMemoryObject.cc (7.3.4.5) - Contains definitions of the `MemoryObjects` that represent pre-mapped kernel memory.

Memory/FaultHandler.cc - Base class for a user-defined page fault handler, called by Domain::basicRepairFault().

Memory/PremappedMemoryObjectCache.cc (7.3.4.5) - The `MemoryObjectCache` used by the `PhysicallyResidentMemoryObject` which maps the portions of the kernel memory that are not readable and the portions that are read-only.

Memory/VirtualMemoryRange.cc (7.3.4.3) - This is the class that describes ranges of virtual memory addresses to initialize the ADDRESS_TRANSLATION class. It was extended to include a ring number that the address range belongs to.

ProcessorDependent/<Processor>/<Processor>CPU.cc (7.3.3, 7.3.4.3) - This is the processor-dependent representation of the processor itself. A setRingRange() method was added to initialize any support of rings that the CPU might do; this is used on the 80386 but not on the NS32332.

ProcessorDependent/<Processor>/<Processor>Context.cc (7.3.3, 7.3.5) - This is the processor-dependent representation of process context. Contains methods for managing switches between rings, particularly saveUserStack(), setNewRing(), and restoreRing().

ProcessorDependent/<Processor>/<Processor>ContextSwitching.s (7.3) - Contains the lowest-level portion of the proxy trap handler.

ProcessorDependent/<Processor>/<Processor>MMU.cc (7.3.3) - This is the processor-dependent representation of the memory-management unit. The basicActivate() method was extended to accept a pointer to the process context from which it determines the ring the process is running in and passes that information to the translation.

ProcessorDependent/<Processor>/<Processor>asmdefs.cc - Source for the file from which structure-element offset defines are created by the 'makhead' tool. These defines go into Configure/System/<Machine>/<Processor>asmdefs.h and are included by <Processor>ContextSwitching.s to access elements in structures. ObjectProxy.h is included here in <Processor>asmdefs.cc so defines for support of proxies will be generated.

ProcessorDependent/<Processor>/ObjectProxyStubs.s (7.3) - The stubs that proxy method calls execute. These stubs are in the kernel but readable by applications.

Tools/Misc/Constcollect.sh (7.2) - Collects constructor names out of a C++ source file by compiling the file and looking in the symbol table for names that are of the form of constructors.

Appendix B

Output Files

This appendix lists files automatically generated by Proxify++ or the make process, including a description of each file. These files are generated for each module that includes Configure/-System/ProxifyCommon.mk.

All files except the alternate header files are in the Configure/$(MODULENAME)/$(MACHINE) directories, where $(MODULENAME) is "System" or "LoadableExample" and $(MACHINE) is "Att6386" or "Multimax".

If the file is discussed in more detail in another section, the section number is in parentheses after the filename.

Alternate header files (7.1.2) - For every header file that is proxified, another header file with the implementation details removed is generated in the Includes/$(MODULENAME)/-$(MACHINE) directory. These alternate header files are included by applications

AllIncludes.cc (7.2) - A list of header files for which alternate header files should be kept. Used by ProxifyCommon.mk after Proxify++ is run. Generated from the $(ALLINCLUDES) make variable.

CONSTRUCTORINSTALLERS.cc (7.3.2) - A two-line file that includes ConstructorInstaller.h and CONSTRUCTORS.h. This is compiled and linked with the proxied module (that is, the kernel or 'load1') to define the proxied constructors that are made available. This file never changes but I thought it was easier to generate it from the makefile than to add another source file somewhere with these two lines in it.

CONSTRUCTORS.cc (7.1.2) - Output of Proxify++ with stubs for all constructors that were marked with the **proxiable** keyword. Used to generate CONSTRUCTORS.h.

CONSTRUCTORS.h (7.3.2) - Output of 'Constcollect' run on CONSTRUCTORS.cc. Contains an invocation of the CONSTRUCT() macro for each constructor. Included in CONSTRUCTORINSTALLERS.cc which is linked with the proxied module and Constructors.s which is linked with applications.

$(MODULENAME)STUBS.cc (7.1.2, 7.3.2) - Method stubs for use when a application-defined method "inherits" from a proxied class. This filename contains $(MODULENAME) because of the g++ compiler which does not allow linking in two files with the same source name together. This is linked with applications.

$(MODULENAME)VERSIONS.cc (7.9) - Contains an invocation of the VERSION() macro for every proxied class including a string that is the output of the 'date' command. This filename contains $(MODULENAME) because of the g++ compiler which does not allow linking in two files with the same source name together. This is linked with applications.

NEWCONSTRUCTORS.h - The "new" copy of CONSTRUCTORS.h. The "new" files are the files that are first generated; to prevent unnecessary recompilations the corresponding non-'new' file is only updated if the file changes.

NEWSTUBS.cc - The "new" copy of $(MODULENAME)STUBS.cc.

NEWVERSIONS.cc - The "new" copy of $(MODULENAME)VERSIONS.cc.

NewAllIncludes.cc - The "new" copy of AllIncludes.cc.

NewProxyIncludes.cc - The "new" copy of ProxyIncludes.cc.

NewTableOnlyProxyIncludes.cc - The "new" copy of TableOnlyProxyIncludes.cc.

PROVIDEDVERSIONS.cc (7.9) - Identical to $(MODULENAME)VERSIONS.cc except invokes PROVIDEDVERSION() macro instead of VERSION() macro. Linked with the proxied module.

PROXYTABLES.cc (7.1.2) - Output of Proxify++ that contains the proxy tables. Linked with the proxied module.

ProxyIncludes.cc (7.2) - The list of header files to be proxified. Used as the main input to Proxify++. Generated from the $(PROXYINCLUDES) make variable.

TableOnlyProxyIncludes.cc (7.2) - The list of header files to be proxified with proxy table only. Included from ProxyIncludes.cc. Generated from the $(TABLEONLYPROXYINCLUDES) make variable.

TableOnlyProxyVersion.cc (7.9) - Contains an invocation of the VERSION() macro which is updated whenever any of the files listed in the $(TABLEONLYPROXYINCLUDES) make variable change. This is only generated for the "System" $(MODULENAME), although perhaps instead there should be one generated for each module with the version string name including $(MODULENAME). This contents of this file is copied into NEWVERSIONS.cc so it is used by both the proxied module and applications.

Bibliography

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–58, January 1985.

[BA+89] J. M. Bernabeu-Auban et al. The architecture of Ra: a kernel for Clouds. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 936–945 vol. 2. IEEE, January 1989.

[Bou83] S. R. Bourne. *The UNIX System.* Addison-Wesley, Reading, Massachusetts, 1983.

[BS88] L. Bic and A. C. Shaw. *The Logical Design of Operating Systems*, chapter 7. Prentice Hall, Englewood Cliffs, NJ, 1988.

[CJMR89] R. H. Campbell, G. M. Johnston, P. W. Madany, and V. F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[CJR87] Roy H. Campbell, Gary M. Johnston, and Vincent F. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3), July 1987.

[CRJ87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, November 1987.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF. In *Proceedings of OOPSLA '89*, pages 49–70. ACM SIGPLAN, October 1989.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128–140, October 1983.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, Reading, Massachusetts, 1984.

[DLA88] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The Clouds Distributed Operating System: functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, June 1988.

[DSS90] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, April 1990.

[Fer89] Jacques Ferber. Computational Reflection in Class based Object Oriented Languages. In *Proceedings of OOPSLA '89*, pages 317–326. ACM SIGPLAN, September 1989.

[GM89] Paulo Guedes and José Alves Marques. Operating System Support for an Object-Oriented Environment. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 37–42. IEEE Computer Society, September 1989.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[HM90] Sabine Habert and Laurence Mosseri. COOL: Kernel Support for Object-Oriented Environments. In *Proceedings of ECOOP/OOPSLA '90*, pages 269–277. ACM SIGPLAN, October 1990.

[IL90] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, April 1990.

[Jor90] D. Jordan. Implementation Benefits of C++ Language Mechanisms. *Communications of the ACM*, 33(9):61–64, September 1990.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing Object-Oriented Designs. Technical Report UIUCDCS-R-91-1696, University of Illinois at Urbana-Champaign, May 1991.

[JZ91] Ralph E. Johnson and Jonathon M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(7):31–34, November 1991.

[KB84] L. J. Kenah and S. F. Bate. *VAX/VMS Internals and Data Structures.* Digital Press, Bedford, Massachusetts, 1984.

[Kee88] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, Massachusetts, 1988.

[KMV+90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object-Oriented Programming*, 3(3):11–22, September 1990.

[Kou91] Panos Kougiouris. Devices Drivers for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, October 1991.

[Lev84] H. M. Levy. *Capability-based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior. In *Proceedings of OOPSLA '86*, pages 214–223. ACM SIGPLAN, September 1986.

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of OOPSLA '87*, pages 17–34. ACM SIGPLAN, October 1987.

[LSU87] Henry Lieberman, Lynn Andrea Stein, and David Ungar. Of Types and Prototypes: The Treaty of Orlando. In *Addendum to the Proceedings of OOPSLA '87*, pages 43–44. ACM SIGPLAN, October 1987.

[Mad92] Peter W. Madany. *An Object-Oriented Framework for Filesystems.* PhD thesis, University of Illinois at Urbana-Champaign, May 1992.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA '87*, pages 147–155. ACM SIGPLAN, October 1987.

[MCK91] Peter W. Madany, Roy H. Campbell, and Panogiotis Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[MCRL89] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[MG89] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122. ACM SIGPLAN, September 1989.

[MLRC88] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, pages 65–79, October 1988.

[MvRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tannenbaum, Robbert van Renesse, and Hans Staveren. Amoeba: A Distributed Operating System for the 1990s. *Computer*, pages 44–53, May 1990.

[Nie89] Oscar Nierstrasz. A Survey of Object-Oriented Concepts. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3–22. Addison-Wesley, Reading, Massachusetts, 1989.

[Not87] David Notkin. Proxies: A Software Structure for Accommodating Heterogeneity. *Software Practice and Experience*, 20(4):357–364, April 1987.

[Org80] E. I. Organick. *The Multics System*, chapter 4. MIT Press, Cambridge, Massachusetts, 1980.

[PD88] D. V. Pitts and P. Dasgupta. Object Memory and Storage Management in the Clouds Kernel. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17. IEEE, June 1988.

[PJC+90] Fred Pollack, Dave Johnson, Dave Carson, Ron Ebrsole, Vittal Kini, Konrad Lai, Bernie Silvernail, and Steve Stacey. A VLSI-Intensive Fault Tolerant Computer Architecture. In *Proceedings of IEE Spring 1990 Computer Conference*, February 1990.

[PKD+90] Fred Pollack, Kevin Kahn, T. Don Dennis, Gerald Holzhammer, Herman D'Hooge, and Steve Tolopka. An Object-Oriented Distributed Operating System. In *Proceedings of IEE Spring 1990 Computer Conference*, February 1990.

[PS85] Jim Peterson and Avi Silberschatz. *Operating System Concepts*, chapter 11. Addison-Wesley, Reading, Massachusetts, 1985.

[R+89] R. Rashid et al. Mach: a system software kernel. In *Digest of Papers: COMPCON Spring 89*, pages 176–178, 1989.

[RC89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278. ACM SIGPLAN, September 1989.

[RJC88] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of OOPSLA '88*. ACM SIGPLAN, 1988.

[RMC90] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, April 1990.

[Rus91] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, January 1991.

[Sel90] Robert Seliger. Extended C++. In *Proceedings of the USENIX C++ Conference*, pages 241–264, April 1990.

[SGH+89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system – assessment and perspectives. *Computing Systems*, 2(7), 1989.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 191–204, Nottingham, UK, July 1989. Cambridge University Press.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM SIGPLAN, September 1986.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[TM86] A. S. Tanenbaum and S. J. Mullender. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–300, 1986.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*, pages 227–242. ACM SIGPLAN, October 1987.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Programming. *Communications of the ACM*, 33(9):104–124, September 1990.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pages 168–183. ACM SIGPLAN, October 1987.

[Wul81] W. A. Wulf. *HYDRA/C.mmp; an experimental computer system.* McGraw, 1981.

[YTM+91] Yasuhiko Yokote, Fumino Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. *Operating Systems Review*, 25(2):22–46, April 1991.

[YTT89] Yasuhiko Yokote, Fumino Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 89–108, Nottingham, UK, July 1989. Cambridge University Press.

Vita

David Wayne Dykstra was born on April 29, 1961, and grew up in Lansing, Illinois. He attended Lansing Christian Grade School and Illiana Christian High School. He attended Trinity Christian College in Palos Heights, Illinois for two years and then transferred to the University of Illinois in Urbana-Champaign where he received the B.S. degree in Computer Engineering in 1983.

David worked part-time for the IIT Research Institute while at Trinity, and at the Coordinated Science Laboratory and Materials Research Laboratory while at the University of Illinois. He worked full-time for AT&T Teletype in Skokie, Illinois, from 1983 through 1987 where he developed software for embedded control of computer terminals and where he developed tools to assist in the software development process. During that same time period he also earned the M.S. degree in Computer Science from the Illinois Institute of Technology in Chicago, Illinois.

David started his Ph.D. work at the University of Illinois in Urbana-Champaign in 1988 and continued to remotely work part time for AT&T Teletype and later for Memorex-Telex Corporation. At the University of Illinois he worked in the Systems Research Group where he did the first port of the *Choices* operating system and worked on extending object-oriented hierarchies beyond the kernel. David is currently a researcher at AT&T Bell Laboratories in Naperville, Illinois, working on an object-oriented telephone switch controller using the CLOS language.

David is a University of Illinois James Scholar and is a member of the ACM and the Tau Beta Pi National Honor Society. His hobby is Amateur Radio. He currently resides in Batavia, Illinois, and is a member of the Wheaton Christian Reformed Church in Wheaton, Illinois.